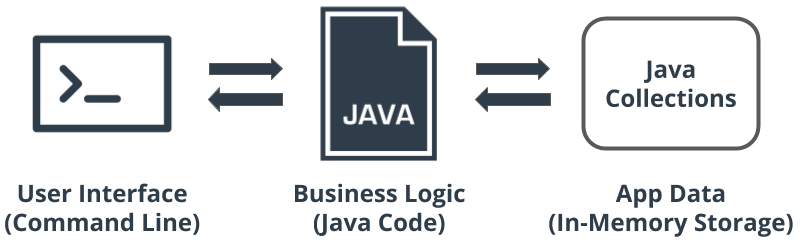
# JAVA

Hotel reservation project:

Application Architecture

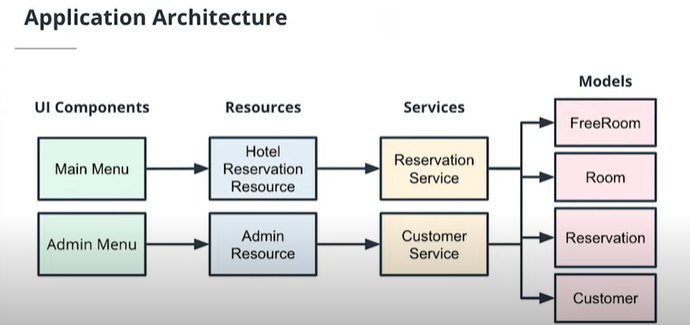
Let's talk about the structure or architecture of the application. The app will be separated into the following layers:

**User interface (UI)**, including a main menu for the users who want to book a room, and an admin menu for administrative functions.

**Resources** will act as our Application Programming Interface (API) to our UI.

**Services** will communicate with our resources, and each other, to build the business logic necessary to provide feedback to our UI.

**Data models** will be used to represent the domain that we're using within the system (e.g., rooms, reservations, and customers).



Layers

An important thing to notice about this architecture is how we use **layers** to support modularization and decoupling. For example, If we later decided to change our UI components to a webpage instead of a command-line interface, layering would support this.

***Layering****is achieved by ensuring there are no cross-communication calls from one layer to another.*

For example, a UI component should never communicate directly with a service. This would expose the service implementation to the UI and make it difficult for us to change it out later.

What is Java and Why Should I Learn it?

Why Learn Java?

There are a lot of good reasons to pursue a career as a Java developer, including:

**Marketability.** Java is frequently [ranked as one of the top programming languages](https://www.indeed.com/career-advice/career-development/best-programming-languages-to-learn), and there are typically tens of thousands of open roles for developers with Java experience at any given time.

**Compensation.** Salary depends significantly on location and level of experience, but in the US, [Java developers average over $100,000 a year](https://towardsdatascience.com/top-10-in-demand-programming-languages-to-learn-in-2020-4462eb7d8d3e).

**Job satisfaction.** In a [ranking of the top 100 best jobs by US News and World Report](https://money.usnews.com/careers/best-jobs/rankings/the-100-best-jobs), software developer was ranked the number one best job in the US. (This isn't a reason to learn Java specifically, but it's a good reason to learn one of the top languages!)

Java is Everywhere

Although it's not always obvious, Java underlies a huge number of the most popular applications we use for everything from online banking to computer games. If you'd like to see some examples, check out [this blog post from Oracle on the 25 greatest Java apps ever written](https://blogs.oracle.com/javamagazine/the-top-25-greatest-java-apps-ever-written).

What is Java?

OK, so Java is really popular and widely used—but what is it? Why was it developed and what sets it apart from some of the other languages that came before it?

Java is an Object-Oriented Language

First, Java is an object-oriented programming language.

*In****Object-Oriented Programming****, we model the components of our software much as if they were real-world objects.*

This allows us to interact with, and develop, our code in a more intuitive way, and it also makes it easier to maintain and re-use the components of our software. If you have a little experience with objects from a previous language you've worked with, but aren't quite sure you understand them, don't worry—we will get into how to create and use objects later in the course.

Java Uses Static Typing

Java uses something called static typing.

*In****static typing****, all variable have a data type that is defined when we first declare the variable, and this type cannot be changed later.*

Static typing means the data type goes with the variable instead of the value that the variable holds. In contrast, dynamic typing is used by some other popular languages (like Python). We will get into this distinction more as we go through the course.

Java is Portable

Finally, a key characteristic of Java is that it is highly **portable**, meaning that Java can be used on many different devices, regardless of the device type (e.g., mobile or desktop) or operating system (e.g., Windows or Mac).

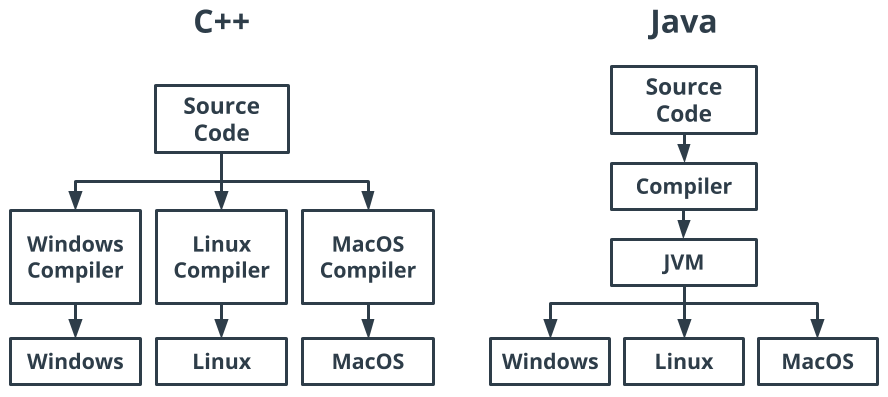
Write Once, Run Anywhere (WORA)

Java was initially developed by James Gosling at Sun Microsystems, which was since acquired by Oracle. Before Java, languages like C and C++ dominated the industry. When Java was first implemented it had the motto **Write Once, Run Anywhere (WORA)**. We can get some insight into what that means by comparing Java with C++.

Suppose we want to write a program that will run on multiple operating systems, such as Windows, Linux, and MacOS. To do this, we have to take the C++ source code that we wrote the program in, and run it through a compiler.

*To****compile****software means to transform the****source code****that we use to write the program into a****machine language****that a computer can understand and execute.*

As shown in the diagram below (on the left), C++ code has to be compiled separately for each operating system, requiring us to create compilers for Windows, Linux, and Mac. This is a lot of extra work.



In contrast, the Java compiler compiles the software for the **Java virtual machine (or JVM)**. The JVM is installed on the users computer and can be downloaded for free—in fact, most computers sold today already come with Java installed. This means we can compile the code once and then run it on every operating system. How does Java do this? Well, instead of compiling code directly into machine code, like C++, Java compiles the source code first into *bytecode*. **Bytecode** is a special language that the JVM can read and transform into machine language. This allows Java to run any Java program on any machine that has a JVM installed on it, regardless of the OS.

### JRE vs JKD

The output of java --version should include **JDK**, which stands for **Java Development Kit**. If the output does not have "JDK" in it, that means you probably only have the **Java Runtime Environment** (or **JRE**) installed. The JRE is needed to run Java programs, while the JDK is needed to compile Java programs. You will definitely will need to be able to compile your programs,

### When to Use Java

In short, Java is especially well-suited for:

* Business applications (client and server)
* Web applications
* Mobile applications on Android devices

But remember, Java really is used for a huge variety of applications. It is a good choice when you need:

* Scalability
* Performance
* Multi-threading
* Memory management
* Portability

***Keywords****are simply words that have a predefined meaning in the Java language.*

Java has 51 keywords, as shown in the table below.

|  |  | **Java Keywords** |  |  |
| --- | --- | --- | --- | --- |
| abstract | continue | for | new | super |
| assert | default | goto | null | switch |
| boolean | do | if | package | synchronized |
| break | double | implements | private | this |
| byte | else | import | protected | throw |
| case | enum | instanceof | public | throws |
| catch | extends | int | return | transient |
| char | final | interface | short | try |
| class | finally | long | static | void |
| const | float | native | strictfp | volatile |
| while |  |  |  |  |

They are reserved words in the Java Language

They cannot be used as identifiers (e.g., when naming variables, classes, or methods)

There are 51 keywords in Java

Here are the key points to remember about **static typing**:

* The data type is bound to the *variable* when the variable is first declared.
* The data type is checked *when the code is compiled*.
* The data type for a variable cannot later be changed.

In contrast, some languages are *dynamically typed.* In **dynamic typing**:

* The data type is bound to the *value* itself, but not to the variable.
* The type is checked during *runtime*.
* The data type of a variable *can* be changed after it is declared; since the type is associated with the *value*, assigning a new value may mean changing the data type.

### what is run time and compile time

Compile time is the period when the programming code (such as C#, Java, C, Python) is converted to the machine code (i.e. binary code). Runtime is the period of time when a program is running and generally occurs after compile time.

## Primitive Values vs Reference Values

In Java, there are two general kinds of values we can assign to a variable:

* A **primitive value** is simply a value, by itself, with no additional data.
* A **reference value** is a value that refers to an object stored in another location in memory.

Objects bundle the primitive value up with additional useful information and behavior. We'll get into how to create and use objects more later on. In Java, there are eight primitive types, as you can see here:

| **Data Type** | **Size** |
| --- | --- |
| byte | 1 byte |
| short | 2 bytes |
| int | 4 bytes |
| long | 8 bytes |
| float | 4 bytes |
| double | 8 bytes |
| boolean | 1 byte |
| char | 2 bytes |

Notice that all of the keywords for primitives start with a lowercase letter.

## Object Reference Types

As we said above, reference types create a reference to a data object. This object can be a Java defined type, like String or Array or it can be a customized, user-defined object. This allows literally infinite flexibility, since you can define whatever object types you need for your particular application.

Unlike primitive types, object types do not have a specific memory allocation size. The *reference* to the object can be of a known, fixed size, but the object itself may vary greatly in size (e.g., the string "hi" will be allocated less space than the string "abcdefghijklmnopqrstuvwxyz").

## Type Casting

**Type casting** is changing one type into another type. There are two kinds of type casting: **Automatic** and **manual**.

## Automatic Type Casting

**Automatic** type casting converts a smaller type into a larger type. For example:

**int** intNumber = 3;

**double** doubleNumber = intNumber;

System.out.println(doubleNumber);

When we print doubleNumber, the value will be 3.0. Notice that there is no precision lost going from a smaller type into a larger type. We started with 3 and ended up with 3.0.

## Manual Type Casting

**Manual** type casting is necessary when we want to do either of these things:

* Convert a larger type into a smaller type
* Convert one object type into another

For example, here we are converting from a larger type (double) to a smaller type (int):

**double** doubleNumber = 3.5;

**int** intNumber = (**int**)doubleNumber;

System.out.println(intNumber);

The resulting value will be 3, not 3.5.

## Truncation

When we go from a larger type into a smaller type, precision is lost. Java cuts off additional data that will not fit in the casted type. So when we go from a double to an int, any values that are not integers will be removed. This is called **truncation**.

Truncation occurs when doing **manual type casting** to convert a **larger type to a smaller type**.

**Truncation** is a loss of precision when going from one to type to another. Basically, we are cutting off or "truncating" the additional data.

Note that this is *not* the same thing as *rounding*. For example, if we round 3.9, we get 4.0. But if we truncate 3.9, we get simply 3! Truncation simply removes the additional data that will not fit, which is why it results in a loss of precision.

## Method Syntax

Methods have four parts:

* **Name.** The name of the method, which we use when calling or invoking the method.
* **Parameters.** The variables that we pass the values to when we call the method.
* **Method body.** The chunk of code, contained within curly braces, { }, that gets run when the method is invoked.
* **Return type.** The data type of the value that the method returns.

## Methods vs Functions

You'll often see the words **function** and **method** used interchangeably. Although the terms are sometimes used loosely, here's the distinction:

* A function is any block of reusable/callable code.
* A method is a block of reusable/callable code that is attached to a class or object.

So a method is a type of function, but it's one that is associated with a class or object, whereas other functions can be executed from anywhere.

In other words, we call any block of reusable code a function, whereas only some functions are also methods. All methods are functions but not all functions are methods.

## Stack vs Heap

Java uses two different memory regions when running an application: The **stack** and the **heap**.

* The stack is used to store primitives and object references, while the heap is used to store the objects themselves.
* Items in the stack get added and removed as a given method executes, while objects in the heap stay until the application is done (or at least, until there are no object references using them from anywhere in the program, at which point they are removed by the garbage collector).
* Items are removed from the stack in a [**Last-In-First-Out (LIFO)**](https://en.wikipedia.org/wiki/Stack_(abstract_data_type))order, meaning that the last element you added to the stack is the first that gets popped off the stack.
* Remember that the items in a stack are only maintained as long as the related method is running. By the time a given method has finished running, all of the items on the stack for that method will have been removed.
* Objects in the heap are accessible from anywhere in the program, while items on a given stack can only be accessed by the related method.
* Stores primitive types and object references=Stack
* Stores Objects.=Heap
* Removes data in a LIFO manner when data is no longer needed.=Stack
* Global memory where data can be accessed from anywhere in the program.=Heap

Access modifiers determine how other classes are allowed to access your variables and methods.

## Types of Access Modifiers

There are four types of access modifiers in Java:

* **Public** means the class can be accessed from everywhere. If you have a method on a class that you want to expose to all other classes, then use this access modifier.
* **Private** means only the defining class can access the data. This provides security, by not allowing other classes to change the data directly. Instead, they must make changes to the data via the provided methods only.
* **Protected** means that access is restricted to the defining class, package, or subclass. This will be useful when we get into subclasses and inheritance in a later lesson, as it will allow our subclasses to use variables and methods from the parent class.
* **Default** means access is restricted to the defining class or the package. This can be used when we have classes inside the same package that we may want to expose data and methods too.

Here's a summary in a table format for your reference:

| **Access** | **Inside class** | **Inside package** | **Outside package by subclass** | **Outside package** |
| --- | --- | --- | --- | --- |
| **Private** | Yes | No | No | No |
| **Default** | Yes | Yes | No | No |
| **Protected** | Yes | Yes | Yes | No |
| **Public** | Yes | Yes | Yes | Yes |

Arrays:

An**array** is a fixed-sized data structure that is used to store multiple values—such as a series of phone numbers, as we saw in the video.

**int** [] numbers = {1, 2, 3, 4};

arrays start with an index of 0, not 1 (this is called [zero-based indexing](https://en.wikipedia.org/wiki/Zero-based_numbering) and is common in programming).

**int** [] numbers = **new** **int**[4];

numbers[0] = 1;

numbers[1] = 2;

numbers[2] = 3;

numbers[3] = 4;

numbers[5] =5=🡺 arrayIndexOutofBoundException

In Java there are three different types of loops:

* while loops
* for loops
* do while loops

Do while loop is guaranteed to execute at least once

**do** {

Execution **block**

} **while**(condition);

And here's a concrete example:

**int** i = 0;

**do** {

System.out.println(i);

i++;

} **while**(i < 5);

## What is JavaDoc?

**JavaDoc** is a documentation generator that produces a searchable HTML document defining the classes and interfaces of an application. This makes it easy for you and other developers to understand the API of an application.

### JavaDoc Comments

The JavaDoc tool reads through Java files and parses certain parts of the code to automatically generate useful documentation. One part of the code that will be picked up by JavaDoc is a **JavaDoc comment**(or simply **doc comment**)**.**

JavaDoc comments are typically added:

* At the top of a class, right before the class name
* For each method in a class We'll get some practice with this when we start defining classes later in the course.

### Syntax

JavaDoc comments use a simple syntax that supports multi-line HTML format documentation. Here's what the syntax looks like:

*/\*\* documentation \*/*

And here's an example:

/\*\* This program HelloWorld produces a standard output

\* displaying "Hello World"

\*

\* @author The author of the **class**

\* @see A reference to another **class**

\*/

## Parts of a JavaDoc Comment

Notice that JavaDoc comments are broken down into two parts:

1. The description
2. Block tags

In the above example, the description is the first part of the comment, and the block tags are the last part (@author and @see).

In this example, we have a doc comment for a method:

/\*\*

\* This method displays a simple **text** output **to** a provided name

\*

\*

\* @param name The name **of** **the** person we want **to** say “Hi” too

\* @return results Returns true **if** **the** name was printed **or**

\* false **if** **it** failed

\*/

Again, the first part is the **description** (in this case, describing the main behavior of the method), and the second part is made up of the **block tags**. In this case, the block tags describe the parameter of the method and what the method returns.

### QUIZ QUESTION

Which of the following statements is correct?

* The JavaDoc tool reads through Java files and parses certain parts of the code to automatically generate useful documentation.
* **Keywords.** If we want to build an application in Java, we first must understand the basic vocabulary of the language. Java's *keywords* will provide us with that core, predefined words of the Java language.
* **Variables.** *Variables* provide us with a way to store data for our application. We'll learn how to use variables in Java and then we'll consider some key ways in which Java variables differ from those of other languages (such as Python).
* **Loops.** *Loops* allow us to iterate over data structures, executing the same code repeatedly until a condition is met, making it possible to process large data structures with only a few lines of code.
* **Methods.** The central idea behind *methods* is that we want to write a block of code once that we we can then re-use many times.
* **Access Modifiers.** We learned how to use *access modifiers* to restrict access to variables and methods, so that users of our applications cannot access all of the data directly and in an unsafe manner. We use them in java to support encapsulation.
* **Javadoc.** How do we know how to communicate with an existing Java application? Or, once we create our own application, how do we share it with others? JavaDoc produces a searchable HTML document that defines the classes and interfaces of an application, making it easy for you or any developer to understand how to use the code.
* **Arrays.** *Arrays* provide us with a way to easily and efficiently store and retrieve collections of data, such as a list of phone numbers.

### Classes Versus Objects

A **class** is like a blueprint for a kind of object. The class defines the *state* and *behavior* that an object of that class will have.

A single class can **instantiate** (or *create*) multiple objects of the same type. So, objects can also be referred to as instances of the class. Each instance of a class is unique, though they all follow the same blueprint defined in the class.

When we **instantiate** (or create) objects from that class, they will all have the states and behaviors defined in the class. Each object is considered an **instance** of the class. Like the dogs in the example, each object can have its own unique state, but all objects will share the same set of behaviors.

## Object-Oriented Programming (OOP)

One of the key characteristics of Java is is that it applies an **object-oriented approach**. This may be different from other languages you may be familiar with.

Benefits of an object-oriented programming approach include:

Better software reusability

Better maintainability

Reduced cost of developing software

What is an Object?

So what is an **object**? Essentially, it is a data structure that we create that bundles together and encapsulates two key things:

The **state** of the object (represented by variables)

The **behavior** of the object (modeled with methods)

This allows us to interact with the components of our application in more intuitive ways, and it also helps protect the methods and variables from being changed in undesirable ways from elsewhere in our code.

Example: Classroom Object

Object-oriented programming can be confusing at first, but once you see enough examples and work with enough objects yourself, it becomes much more intuitive. Let's have a look at another example, and start to get an idea of what an object looks like in code.

### Defining a Class

Parts of a Class Definition

Classes are composed of the following different parts:

The **class name**, which is always formatted in [UpperCamelCase](https://en.wikipedia.org/wiki/Camel_case" \t "_blank).

The **class variables**, which can be either primitives or object references. Note that you'll want to mark these private to protect your instance variables so other classes do not have direct access to them.

The **constructor** which is a special method used to initialize the object when it is created.

The **methods** used to define the behavior that objects of the class will share.

**Accessor methods** provide access to instance variables. The names of accessor methods typically start with "get".

**Mutator methods** allow other objects to change the values. The names of mutator methods start with "set".

### Instance Variables vs Class Variables

Note the difference between instance variables and class variables.

**Instance Variables** are state variables that can have unique values for each object.

**Class Variables** are state variables that belong to the class itself, and are the same for every object. The static keyword identifies this variable as belonging to the class (not to individual objects).

Example

For your reference, here's the example code we looked at in the video:

**public** **class** **Student** {

**private** **final** String id;

**private** **final** String firstName;

**private** **final** String lastName;

**public** **Student**(String id, String firstName,String lastName){

**this**.id = id;

**this**.firstName = firstName;

**this**.lastName = lastName;

}

**public** String **getId**(){

**return** id;

}

**public** String **getFirstName**(){

**return** firstName;

}

**public** String **getLastName**(){

**return** lastName;

}

**public** **void** **setStudent**(String id, String firstName,String lastName){

**this**.id = id;

**this**.firstName = firstName;

**this**.lastName = lastName;

}

}

**QUIZ QUESTION**

Below are the different parts of a class we just discussed. Can you match each part with the correct description?

 These are the correct matches.

Description

Part of the class

A type of function used to model the behavior of the class.

Methods

Used to initialize objects during instantiation.

Class constructor

Used to provide state for the objects.

Instance and class variables

Used to identify not only the class itself, but also the associated Java file.

Class name

Final Keyword

The final keyword is a non-access modifier. This keyword is used for classes, methods, and variables. The keyword makes items non-changeable. This means they cannot be changed and cannot be overridden or inherited. The final keyword is used when we don’t want other developers to change or modify a class state.

# Working with Objects

As we've covered before, one of Java's key characteristics is that it is *object-oriented*. This means operations are performed on objects and not directly on variables. In this paradigm, we start by creating our classes first, which contain the behavioral logic necessary to perform our operations.

We define the class by:

* Giving it a name
* Setting up the instance variables (states)
* Initializing our instance variables within the constructor
* Defining the methods (behavior)

There will be times when we need to make an object a**Singleton class**. Which means that there will only be one instance of the class in the JVM. We use singleton class for utility classes and services when we only want one instance. This is to ensure that the state data or methods are shared with all of the other objects in the JVM. In this example, we are creating a static reference to the class server. To access the server we use the class name with the method getInstance(). This Singleton approach will ensure that every object that references the server class will have access to the server's hotelRoom ArrayList.

public class Server {

private static Server reference = new Server();

private ArrayList hotelRooms;

private Server(){

this.hotelRooms = new ArrayList();

}

public Server getInstance(){

return reference;

}

}

After writing our code, we are ready to test. To run the test, we create a **Driver** or **Tester** class. This is a class that has a **main** **method**. (Remember, the main method in Java is a special method and is the entry point to executing code.)

This can all seem very abstract, so next we will walk through an exercise where you'll get some practice working with objects in Java.

# Garbage Collection

Every new object that we instantiate is added to the heap and consumes memory. In programming languages like C or C++, developers are responsible for creating and destroying objects. In contrast, Java has a background process called **Garbage Collection (GC)** that checks all instantiated objects, and destroys them if they do not have any references—thus freeing up memory, even without any active management on the part of the developer.

## Advance OOP

Packages:

In Java, we use **packages** to organize and manage our Java files:

If you are used to the concept of [namespaces](https://stackoverflow.com/questions/3384204/what-are-namespaces), packages serve essentially the same purpose in Java.

Packages are not unlike the traditional folders that you use every day on your computer for organizing your files.

By nesting packages and files, we can create a hierarchy that allows us to identify specific files and avoid naming conflicts.

The package keyword will be located at the top of a Java file, typically on the first line of the file, before the imports and class name. For example:

package project.src.api

In Java, packages are…

Used to organize and manage Java files.

Used to distinguish Java files with the same name.

## Inheritance

**Inheritance** is one class acquiring properties and methods from another class. Here are some key points you should remember about inheritance:

We want to go from general to specific. The **parent** or **superclass** is the most general and the **child** or **subclass** is the more specific.

By **extending** the superclass you are stating that the subclass is of the superclass type. When we're not sure if a subclass is inheriting from a parent class, we can use the “is a” test (e.g. a car is a vehicle).

The relationship between superclass and subclasses is only one way. The subclasses need to know about the superclass, but the superclass should never know anything about its subclasses.

The Object Superclass

Every class inherits from the superclass Object. Because all objects inherit from the Object class, there are some methods that all objects have, no matter what types they are. For example, all objects have:

clone(), so that we can clone or make a copy of any object.

equals(), which we can use to determine if two objects are the same.

hashCode(), which provides a unique hash code for each object. This is something we'll make use of later on when we need to store and retrieve objects in specific data sets.

toString(), which we can use to get a description of the current state of an object.

QUIZ QUESTION

Why do we use inheritance?

(Select all that apply.)



To share state only and not behavior.

To promote code reusability and reduce duplicate code.

To create relationships between classes, going from general to specific.



To promote code duplication.

To share state and/or behavior with other related classes

Vehicle Example in Code

Now that we have discussed the general concept of inheritance, let's take a look at how the vehicle example might be implemented in actual code. You aren't expected to write this code yourself—this is just an example. We will go through an exercise shortly in which you will get the opportunity to make use of inheritance.

# Abstract Classes

An **abstract class** has the following key characteristics:

It defines the behavior for each of the subclasses, but we cannot directly instantiate the abstract class itself.

It allows us to create **abstract methods**.

An abstract method is a method that does not contain an implementation body. Instead, it simply provides a header for the method.

Subclasses that extend an abstract class are required to override all abstract methods and provide a specific implementation.

Example: Abstract Vehicle Class

Here's the example we looked at in the video. To ensure that we will not be able to directly instantiate Vehicle objects from this class, we define it using the abstract keyword:

**public** **abstract** **class** **Vehicle** {

**protected** String start;

**protected** String stop;

**protected** String direction;

**public** **Vehicle**(String start, String stop,

String direction) {

**this**.start = start;

**this**.stop = stop;

**this**.direction = direction;

}

**public** **abstract** **void** **speed**();

}

Next, we use the Extends keyword to have our Car class extend the Vehicle class:

**public** **class** **Car** **extends** **Vehicle** {

**public** **Car**() {

**super**("Car start", "Car stop", "Car direction");

}

@Override

**public** **void** **speed**() {

System.out.println("55");

}

}

Thus, although we cannot directly instantiate objects from the abstract Vehicle class, we can instantiate objects from the Car class, and they will have all the behavior of a Vehicle.

### Interfaces:

**Interfaces** allow us to avoid hardcoding features in an application. We can move specific implementation details into subclasses, and then use an interface to communicate between the application and the subclasses.

## Interfaces vs Abstract Classes

Here are some of the similarities and differences between abstract classes and interfaces:

**Abstract class**

* Can have class variables.
* Can have both **abstract** methods and **concrete** methods that are shared with the subclasses.
* **Can have instance variables**, i.e. variables that are specific to individual subclasses.
* Subclasses can only extend one class.

**Interfaces**

Can have class variables.

Every method in an interface is **abstract**.

**Cannot have instance variables**. Variables in an interface must be the same for every class implementing the interface.

Classes can implement more than one interface and **have multiple inheritance**.

When to Use an Interface

We use an interface when:

We expect unrelated classes will be implementing our interface.

We want to support multiple inheritance.

We want to specify the behavior for a data type, but we do not care about the implementation.

**QUIZ QUESTION**

For each of the descriptions below, does it better describe an interface or an abstract class?

 These are the correct matches.

Description

Interface or Abstract class

We expect unrelated classes will be implementing our super class.=Interface

We want to support multiple inheritance.=Interface

We want subclasses to have instance variables.=Abstract class

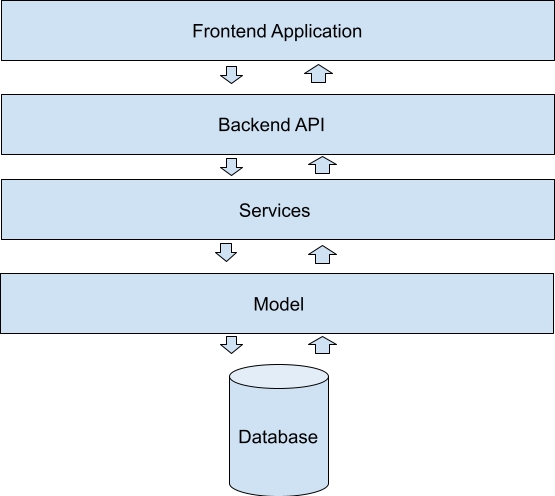
We want to provide concrete methods for subclasses=Abstract class

We want to make use of both concrete methods and abstract methods for subclasses.=Abstract class

# Layering

Now that we have learned about objects, it’s time to learn about another important architectural concept that is used when building an application of objects. We call it layering when only one group of similar objects is allowed to access another group of objects. We typically think of layering as a horizontal group of objects with the same functionality accessing only the objects below them.

In the example below, we can see the frontend application code should only be allowed to access the backend API. The reason we use the layering approach is for security, stability, and testing. The backend API is secure from software attacks and, therefore, should be the only part of the application exposed to the frontend application software. Additionally, the backend API should only access the services of the application and never directly access the data model or the database. This creates a structured environment that allows for easy testing because each of the different objects in an application should only access objects within their layer or directly below them.



In this lesson we learned more about key concepts in object-oriented programming with Java:

**Packages**, which are essentially like folders you can use to organize your code and identify exactly which file (or class) you are referring to.

**Inheritance**, where one class acquires properties and methods from another class.

**Abstract classes**, which cannot be directly instantiated themselves, but that allow us to define the behavior for each of the subclasses.

**Interfaces**, which support *decoupling* and allow us to avoid hardcoding features in an application.

**Polymorphism**, which is the ability for an object to take on many forms.

## Common Types

Exceptions:

The Error Class and the Exception Class

The Java error-handling framework uses two different classes to identify abnormal software events:

The **Error** class is used to indicate a serious problem that the application should not try to handle.

The **Exception** class is used when there is a less catastrophic event that the application should try to handle.

The Throwable Class

Both Error and Exception classes inherit from the abstract class **throwable**. This means that both errors and exceptions will contain:

**The type of problem** – the class type, either Exception or Error

**The problem message** – whatever description you've provided within the class, such as "out of memory"

**The stack trace where the exception occurred** – the order in which things ran and the place in that sequence where the problem occurred (this information is used by developers to track down issues and resolve problems in the code)

Throwing an Exception

What does it mean to throw an exception?

When an error occurs within a method, that method creates an Exception object.

Next, the method hands the Exception off to the JVM runtime system. And that's what we mean when we say that we are throwing an exception. We simply mean that a method hands off an exception to the runtime.

Checked vs Unchecked

In Java there are two different types of exceptions:

Unchecked Exceptions

**Unchecked** exceptions are exceptions that are unknown to the compiler.

Because these exceptions are only known at runtime, they are also referred to as runtime exceptions.

They are a result of a programming error, typically arithmetic errors (such as division by 0).

Unchecked exceptions are used when we expect that the caller of the method cannot recover from the exception.

Checked Exceptions

**Checked** exceptions are known to the compiler.

If we are calling a method that potentially throws a checked exception, it must be handled (or we will get an error from the compiler).

Checked exceptions are used when we expect that the caller of the method can recover from the exception.

QUIZ QUESTION

Which of the following are **true** statements about exceptions and errors?

(Select all that apply.)

The Exception and Error classes inherit from the Throwable class.

Error are thrown for catastrophic events where the application is not expected to recover.

Exceptions are thrown for problems where the application is expected to recover.

Throwing an Exception is when an Exception object is created and handed off to the Java Runtime.

To handle exceptions, we need to write an **exception handler.** This involves three main components:

A try block

A catch block

A finally block

Here's an example:

**try** {

**read**();

}

**catch** (FileNotFoundException ex){

ex.getLocalizedMessage();

}

**finally** {

}

In this example, we first try to call the read method to read a file. If that doesn't work, the catch block throws a FileNotFoundException.

Let's take a close look at each component.

The try Block

The try block contains the code we want to try to run. In the example, we are trying to call the read method:

**try** {

read();

}

The catch Block

A catch block is an exception handler that handles one specific type of Exception. In this example. The type of exception we are handling is a FileNotFoundException exception:

**catch** (FileNotFoundException ex){

ex.getLocalizedMessage();

}

Remember, FileNotFoundException is a class and—as with all exceptions—it inherits from the Throwable class.

Inside the catch block, we add the code we want to execute when the exception is thrown—in this case, we are calling a method called getLocalizedMessage.

The finally Block

The last component of the handler is the finally block. This is an optional block and, in our example, you can see that it is empty:

**finally** {

}

The finally block is always executed–even if an unexpected error causes the method to terminate early.

QUIZ QUESTION

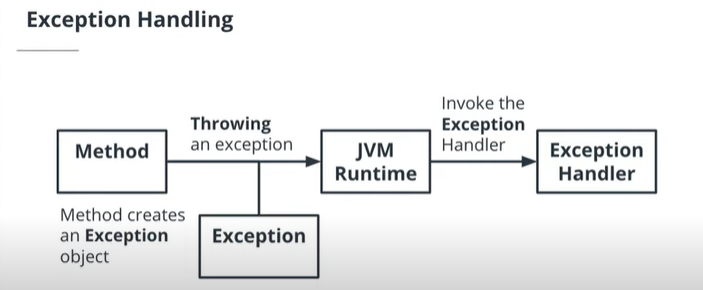
Which of the following are **true** statements about handling exceptions?

(Select all that apply.)

Java uses three different blocks for Exception handlers: try, catch and finally.

Exception handlers handle one specific Exception class type.

Exceptions are handled by creating an Exception Handler.



You can create and throw your own exceptions by extending the Exception classes

There are two types of exceptions in Java: Checked and unchecked. Checked exceptions will get caught at compile time and will not allow the code to build until they are either in a catch block or thrown. Unchecked (or *runtime*) exceptions are not checked by the compiler.

* **public** **class** **Phone** {
* **private** **final** String phoneType;
* **private** **final** String phoneNumber;
* **public** **Phone**(String phoneType, String phoneNumber) {
* **super**();
* **if** (phoneType == **null** || phoneNumber == **null**) {
* **throw** **new** IllegalArgumentException("The type or number cannot be null");
* }
* **this**.phoneType = phoneType;
* **this**.phoneNumber = phoneNumber;
* }
* **public** String **getPhoneType**() {
* **return** phoneType;
* }
* **public** String **getPhoneNumber**() {
* **return** phoneNumber;
* }
* @Override
* **public** String **toString**() {
* **return** phoneType + " " + phoneNumber;
* }
* }
* **public** **class** **PhoneExceptionTester** {
* **public** **static** **void** **main**(String[] args) {
* String[] numbers = **new** String[] { "123-4567", **null**, "234-4567", "345-5678" };
* **for** (**int** i = 0; i < numbers.length; i++) {
* **try** {
* System.out.println(**new** Phone("iPhone", numbers[i]));
* } **catch** (IllegalArgumentException ex) {
* System.out.println(ex.getLocalizedMessage());
* }
* }
* **for** (**int** i = 0; i < numbers.length; i++) {
* System.out.println(**new** Phone("iPhone", numbers[i]));
* }
* }
* }

## What is an Enum?

In many cases when developing software we need to provide a predefined value for a single variable type. **Enumerations** (or **Enums** for short) are a special data type of constants that allow a variable to be set from an enumerated list.

An Enum is a Class

In Java, the declaration of an Enum defines a class. This class can exist **within another class** or as a **standalone class**.

In some cases, we might only need to use the Enum type within a single class (as would probably be the case with the stoplight example). In that case it would be acceptable to define it within another class rather than as a standalone. However, if other classes are expected to use the enum type, it should be defined as a separate, standalone class.

Syntax Example

Below is an example of how to create a stoplight enum like we showed in the video. Notice that we have created three constants, RED, YELLOW, and GREEN. In Java, it is standard to put the values of enums in uppercase to indicate that they are constants.

Creating an Enum

**enum** Stoplight {

RED,

YELLOW,

GREEN

}

Assigning a Variable

Next, we will assign a variable, myStopLight, with the RED enum.

**Stoplight** myStoplight = Stoplight.RED;

Defining an Enum Inside a Class

We can also create an enum from inside a class:

**public** **class** **Main** {

**enum** StopLight {

RED,

YELLOW,

GREEN

}

**public** **static** **void** **main**(String[] args) {

StopLight myStoplight = Stoplight.RED;

System.out.println(myStoplight);

}

}

QUIZ QUESTION

Select the correct statement about enumerations (enums).

Enums are a special data type of constants that allow a variable to be set from an enumerated list.



### Scanner

The Scanner class can read and parse simple text. Here are some key points to keep in mind:

It parses primitive types and String types into tokens.

By default it uses whitespaces to delimitate each word. However, it can also use regular expressions.

The Scanner class can read from several different types of sources, like strings, files and System.in (to get input from the command line).

Scanner Syntax

Example 1

We can use the Scanner class to get input from the command line. To do so, we instantiate a scanner object, passing in System.in:

Scanner scanner = **new** Scanner(System.in);

Example 2

In this example we are using the nextLine method to return the full line of the input:

Scanner scanner = new Scanner("This is a line");

System.out.println(scanner.nextLine());

**Output:**

This is a line

Example 3

In this next example, we are using the next method to read the first token. The next method finds and returns the next complete token.

Scanner scanner = **new** Scanner("This is a line");

System.out.**println**(scanner.**next**());

**Output:**

This

Example 4

In the final example, we are using the hasNext() method in a while loop to determine if it is safe to call the next method. We only want to call the next method when we know there is a token available.

***Note:****By default the Scanner tokenizes input by whitespaces. Let's say we have a string with the following text*"One Two Three"*. The text will be tokenized into three separate tokens,*"One"*,*"Two"*and*"Three"*.*

Scanner scanner = **new** Scanner("This is a line");

**while**(scanner.hasNext()) {

System.out.**println**(scanner.**next**());

}

**Output:**

This

is

a

line

QUIZ QUESTION

Which of the following statements are **true** about the Scanner class?

The Scanner class can be used to read input from the console.

The Scanner class can also use RegEx to parse console input.

SUBMIT

**QUIZ QUESTION**

Match the code with the correct output for the String "1 number";

 These are the correct matches.

Code

Output

scanner.nextLine()1 number

scanner.next() 1

**while**(scanner.hasNext()) {System.out.println(scanner.next());}

1  
number

scanner.nextInt() 1

Additional Resources

Java also has two other classes, BufferedReader and Console, that can be used to read user input from the command line. If you'd like to learn more, you can check out [this article on three ways to read user input from the console](https://www.codejava.net/java-se/file-io/3-ways-for-reading-input-from-the-user-in-the-console).

### Date and Calendar:

The Date Class

The **Date** class represents a specific instance in time. We can instantiate a new Date object like so:

Date date = **new** Date();

The Calendar Class

The **Calendar** class is an abstract class that provides methods for manipulating date and time. The basic syntax for instantiating a new Calendar object looks like this:

Calendar calendar = Calendar.getInstance();

**QUIZ QUESTION**

For each of the descriptions below, does it better describe the Date class or the Calendar class?

Represents only a specific day and time.Date

Represents a specific day and time and provides transformation methods.Calendar

Has methods to set a specific Day, Month and Year.Calendar

Is an abstract classCalendar

Is a concrete classDate

Additional Resources

Java has some other classes—LocalDate, LocalTime, and LocalDateTime—that can be used to manipulate dates. These won't be needed for this course, but if you're curious, you can read about them in the [official documentation here](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html).

RegEx:

**Regular expressions** (often abbreviated as **RegEx**) are used to match or find strings based on a specialized syntax.

The **regEx** package in Java contains three classes to support these operations

Pattern

Matcher

PatternSyntaxException.

To use RegEx in Java, we have to do two main things:

Create a Pattern based on a specialized syntax

Use the Matcher to determine if the pattern exists in the String provided

RegEx Syntax

Helpful RegEx Resources

As a Java developer, do you need to thoroughly learn and memorize RegEx syntax? Not necessarily. Unless you're using it very heavily, it will probably suffice to look up the expression you need when you need it. Along those lines, here are some handy resources that you may want to bookmark for later reference:

For use of the RegEx class in Java, see the [official Java docs on regular expression syntax](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html).

For coming up with the regular expression itself, try playing with [RegExr.com](https://regexr.com/). You can enter some text and then try out different expressions; when there is a match, the text will be highlighted. The page also has a handy cheatsheet for commonly needed RegEx characters.

Syntax Example

Let's look at the example from the video. First, we start by defining a String for a valid email address using a regular expression:

String emailRegex = "^(.+)@(.+).(.+)$"

Next we use **Pattern**, which provides a compiled instance of a String regular expression:

Pattern pattern = Pattern.compile(emailRegex)

Finally, we use **Matcher**, which contains all of the state data for matching a character sequence against the Pattern:

Matcher matcher = pattern.matcher("jeff@example.com")

QUIZ QUESTION

Select the correct regular expression string to enforce a valid email address, such as:

jeff@example.com

If you like, you can try out the patterns on [RegExr.com](https://regexr.com/).

"^(.+)@(.+).(.+)$"

### Generics and Collections:

**Generics** allow us to write methods that handle a whole group of different data types—in other words, to write methods that are generic.

The benefits of using Generics are:

Stronger type checks at compile time.

They remove the need to cast objects.

They allow developers to implement generic algorithms.

Put another way, generics are a way to parameterize class types into classes, methods, and variables.

Syntax Example

Here is some code that creates a list of strings without using generics:

List strings = **new** List();

How do we know what’s in this list? The list is called strings, but that is just the name we chose. We would need to cast the items in order to use them as String types.

With generics, we can identify what types are stored in the list. The syntax for a generic is very simple—it's just bracket with the type inside, < type >. So in this example, we can add <String> right next to List to identify what types are stored in the list:

List<String> strings = **new** <String>ArrayList();

Because we've used generics to identify the items in the list as Strings, we could now call methods directly on them without having to first cast them.

QUIZ QUESTION

Select the correct option for creating a List of Integers using generics.

List<Integer> number = new <Integer>ArrayList();

Collections:

**Collections** are a set of data structures that were introduced in Java 5.0 to solve problems with consistency between data structures and to address performance issues. They were developed with the following goals in mind:

**High performance/efficiency.**

**High degree of interoperability.** The new data structures all needed to behave similarly to each other. In practice, this means that Collections data structures implement similar interfaces and extend similar abstract classes.

**Integrate with existing APIs seamlessly.** The Java framework had to extend and adapt to the new data structures easily. They needed to integrate the new data structures seamlessly with little disruption to the existing API.

Now that we have learned about why collections were created. Let’s take a look at what they are and how to use them in InteliJ.

Syntax Example

The Collections framework consists of several different data structure classes like List, Maps and Queues. In addition, there are utility classes like Collections and Arrays that provide methods for sorting and creating empty lists. Below is an example in which we create a List of type String and then add and remove data.

***Notice:****We are creating an instance of*List*but we are using an*ArrayList*to instantiate it. This is because*List*is an Interface and we must use a concrete class to instantiate the object.*

List<String> myList = **new** ArrayList<String>();

myList.add("one");

myList.add("two");

myList.add("three");

*//We could also remove the element "one" by using the index 0 instead of the value "one".*

myList.remove("one");

QUIZ QUESTION

Which of the statements below are **true** about collections?

(Select all that apply.)

Collection inherits from the Iterable interface.

Map, List, Set and Queue all implement the Collection interface.

Collection is an Interface.

### Sorting Collections

The collections framework provides a sort method that can be used to sort lists containing the following types of objects:

Strings

Wrapper objects

User-defined classes

Let's have a look at the syntax for each of these.

Strings

A list of Strings can be sorted by simply passing the list to the Collections.sort() method. In the example bellow, the names will be sorted in ascending order.

List<String> names = **new** LinkedList<String>();

names.add("Mike");

names.add("Bob");

names.add("Alice");

Collections.sort(names);

Wrapper Objects

Wrapper objects were introduced in Java to wrap primitive variable types into objects. In the example below, the primitive int values are being converted to Integer objects and then sorted.

List<Integer> numbers = **new** LinkedList<Integer>();

numbers.add(201);

numbers.add(100);

numbers.add(101);

Collections.sort(numbers);

User-Defined Classes

User-defined classes will need to implement the **Comparable** Interface in order to use the Collections.sort() method. The Comparable Interface provides a method, compareTo, which is used to compare two objects of the same type. In the example below we are using the Person object's name field to compare Person objects. This line of code, name.compareTo(person.name), handles all of the hard work.

**import** java.util.\*;

**class** **Person** **implements** **Comparable**<**Person**> {

**public** String name;

**public** **Person**(String name) {

**this**.name = name;

}

**public** **int** **compareTo**(Person person) {

**return** name.compareTo(person.name);

}

}

**public** **class** **PersonSort** {

**public** **static** **void** **main**(String[] args) {

ArrayList<Person> people = **new** ArrayList<Person>();

people.add(**new** Person("Same"));

people.add(**new** Person("Mike"));

people.add(**new** Person("Apple"));

Collections.sort(people);

**for** (Person person : people) {

System.out.println(person.name);

}

}

}

Note that it is up to us as the developers to select the variables we want to use when we compare two user-defined classes.

QUIZ QUESTION

Select the correct syntax for sorting an ArrayList of Strings called names.

Collections.sort(names);

## What is a Set?

Sometimes, as with the example of a list of phone numbers, it's important for the values in a collection to be unique. Fortunately, we have Sets. A **Set** is a collection type that has no duplicate values.

Note that Set extends the Collection interface and therefore provides the same methods as other Collection data structures do.

Concrete Implementations

There are three concrete implementations for the Set interface:

HashSet

TreeSet

LinkedHashSet

Each of these implementations vary, but all of them enforce the no-duplicate-values requirement of the Set interface.

***Note:****Above, we referred to these as "concrete implementations". Remember, "concrete" means that they are not interfaces or abstract classes, and they contain all of the code necessary per the interface.*

Set Syntax

For your reference, here are the syntax examples we just looked at for creating and using sets.

Creating a Set Object

In the Collections framework, the Set is an interface and cannot be directly used to instantiate a class. In the example below we are creating a Set of strings. Notice, we are using the concrete class HashSet to instantiate our Set object.

Set<String> mySet = **new** HashSet<String>();

Adding to a Set

Next we will look at adding objects to our Set. Remember that a Set has an additional restriction that no duplicate values are allowed. In this example we are adding Strings to our Set using the add method:

mySet.add("Hello");

mySet.add("Hello");

mySet.add("Hellos");

Even though we used add three times, this will actually only add two items, since one of the items is a duplicate.

Retrieving an Object From a Set

In this example we will be looking at the syntax to retrieve a String object from our Set. Unfortunately, the only way retrieve an object from a Set is to iterate over the Set looking for the object. Below we are iterating over the Set to look for the "Hello" String. This example is simple but provides the necessary techniques for finding an object in a Set.

String foundObject;

**for** (String **text** : mySet) {

**if**(**text**.**equals**("Hello")){

foundObject = **text**;

}

}

### Queues

Sometimes we need to process items in a **First-In, First-Out** (**FIFO**) order.

**Queues** are data structures that allow us to process items in a **First-In, First-Out** (**FIFO**) order. The first item placed in a queue is also the first one removed from the queue.

*An everyday example is a line at the grocery store, where the first customer to arrive at the checkout counter should be the first person to leave the store. There are many analogous cases in computer programs, such as when we have a list of processes and need to execute them in the order they arrive.*

Some key points about Queues in Java:

Queue is an interface that implements the Collection interface

We put newly added elements at the end of the queue

We pop elements off the front of the queue

Queues Syntax

For your reference, here are some basic syntax examples for creating and using queues.

Creating a Queue object

In the collections framework the Queue is an interface and cannot be directly used to instantiate a class. In the example below, we are creating a Queue of strings. Notice that we are using the concrete class LinkedList to instantiate our Queue.

Queue<String> myQueue = **new** LinkedList<String>();

Adding to a Queue

Next we will look at adding objects to our Queue. In the example below we are adding Strings to our Queue using the add method.

myQueue.add("Hi");

myQueue.add("There");

Retrieving an Object From a Queue

In this example we will be looking at the syntax to retrieve a String object from our Queue. Remember, Queues are First-In, First-Out (FIFO) data structures. So the first object in the queue will also be the first object out of the queue. We retrieve objects from our queue by using the poll method. This method both retrieves the first element from the Queue and removes it from the Queue.

myQueue.poll()

Iterating Over a Queue

In our final example we will be looking at using the while loop to iterate over our Queue and remove all of the elements. In the example below we are displaying all of the elements that were in the queue.

**while** (!myQueue.isEmpty()) {

System.out.println(myQueue.poll());

}

QUIZ QUESTION

Which of these statements are **true** about Queues?

Select all that apply.)



Queues insert to the end of the Queue and pull from the top of the queue.



Queues follow a First in and First out (FIFO) concept.

### Overriding Hashcode and equals

Now that we have learned about the Collections framework, let’s cover an important topic. What do we do if we want to sort a list of people objects by their addresses? There are several ways we can do this. We can use the Comparator Interface, but If we are using the Collections framework, this can be achieved by overriding the equals and hashcode methods. The equals and hashcode methods play an important role when using the Collections. When we call Collections.sort() it uses the equals method to sort the Collection.

Additionally, the Hashcode is used when inserting objects into a Hash structure. There is a contract we must ensure we meet when overriding the equals and hashcode. If we change one of them, we must always change both.

Quizzes

public class People {

private String address;

private String name;

private String number;

}

QUIZ QUESTION

In the example code above, we have the People object. Select the correct equals method to sort objects by their address.

@Override

**public** **boolean** **equals**(Object o) {

**if** (**this** == o) **return** **true**;

**if** (o == **null** || getClass() != o.getClass()) **return** **false**;

People people = (People) o;

**return** address.equals(people.address);

}

QUIZ QUESTION

In the People class example above. Select the correct example for overriding the hashcode.

@Override

**public** **int** **hashCode**() {

**return** Objects.hash(address);

}