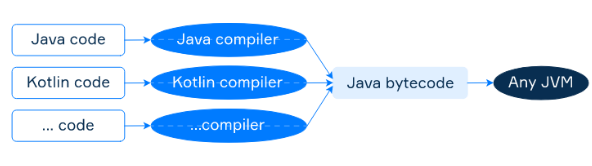
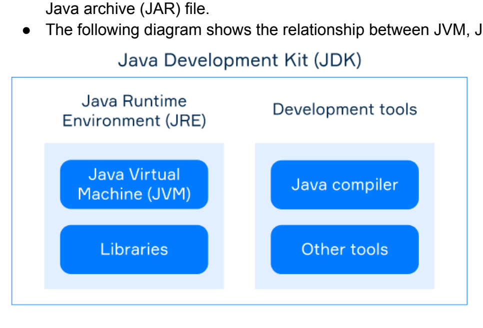
Introduction

- Java is a statically typed language. One of its key features is that it can be run on many different platforms without modifying the code. This features is known as “Write once, run anywhere”
- Java includes a built-in garbage collector that frees memory during runtime
- While Java supports multiple programming paradigms, it is primarily an object oriented language: almost every part of the program can be considered an object. The program itself can be considered as a set of interacting objects.
- When a vanilla Java application is run, a bunch of objects are instantiated corresponding to the classes. These objects are related, and there is thus a network of objects that are connected using object references. This network of objects basically maps out dependencies between objects.
- Values such as numbers and strings are called ‘literals’. Java supports several types of literals
- Public classes should only be declared in a .java file named after the class. Therefore, you can only have one public class per file
- Java source code is written in .java files. A compiler (such as javac) compiles into a much lower level form known as “bytecode”. Bytecode cannot be executed natively by a computer, it is instead used by JVM (Java Virtual Machine). JVM, which can be installed on many runtime environments, converts the bytecode into native instructions and executes them on the computer.
- Bytecode has a .class file extension
- Bytecodes are generic and not specific to any runtime environment. Only the JVM is specific to a system. Therefore, you can have generic Bytecode that can run on Linux JVM, a mobile JVM, etc.
- Because a compiler converts Java source code to bytecodes, compilers also exist that compile non-Java languages to bytecode. The JVM doesn’t know or care what language or environment generated the bytecode. The following illustration paints a picture of this:



- Obviously each platform has its own JVM, but they all behave identically with a given bytecode, since they all function according to the JVM specification document. Java HotSpot is one of the more popular JVMs
- Obviously the compiler runs natively on the system you’re using to write Java code, so the compiler itself isn’t platform independent. However, all compilers regardless of platform generate the same generic bytecode
- The JRE (Java runtime environment) is an execution environment for running bytecode. It includes the JVM and the Java Class Library. Therefore, you almost certainly need a JRE to run your bytecode it contains the class library
- A JDK (Java Development Kit) includes everything in the JRE plus a Java compiler, debugger, etc.
- In practice, Java programs often consist of multiple .class files bundled into a Java archive (JAR) file.
- The following diagram shows the relationship between JVM, JRE, and JDK:



-

Basics

- In Java, we can use underscores in integers to make it more readable. For example, 1_000_000 is equivalent to (and more readable than) 1000000
- Characters are surrounded by single quotes, while strings are surrounded by double quotes. This is important! `'Text'` is not a valid literal since it is surrounded by single quotes (which causes Java to interpret it as a character). But it is not a valid character since it has multiple characters.
- Every Java program needs to have a “public class”. It is the basic unit of the program and every Java application must have at least one class.
- The public class needs to have a main method, which will be the entry point of the program. See the snippet below to illustrate:

```
public class HelloWorldProgram {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

- “Standard Output” is a receiver through which programs can send information as text, it is supported by all operating systems. Java provides a special object called “Systems.out” to work with the standard output. For printing, two of its methods include println and print. They work similarly, except println automatically ends with a newline whereas print does not.
- Variable names and static strings can be combined in a print statement by using the + symbol. For example, `System.out.println("Hello "+name);` where name is a variable. Also, the `\n` escape sequence can be used to print a newline anywhere inside the print statement.
- There is also a method `System.out.printf`. It allows for C style string formatting instead of having to do string concatenation when printing variables combined with text.
- In addition to declaring a specific type when declaring a variable, we can use the `var` keyword to declare a variable whose type will automatically be inferred (similar to auto in C++)
- Variable names can include only the special characters $ and _
- Variables for numeric types include `byte`, `short`, `int`, `long`, `float`, and double. The size of these types are fixed and do not depend on the OS or hardware.
- Other data types include `String` and `char`. It’s important to note that `String` is a class, not a primitive type.
- There exists a class `Integer`, which has static methods that can be useful when working with `int` types. For example, it includes a method `parseInt` which is used to convert a `String` (of numerical characters) to an `int`
- The unary operator - can be used to make an integer variable negative. For example `int temp=10; int temp2=-temp;`
- Java performs arithmetic operations in standard PEMDAS order
- We often need to assign the value of a variable to another variable of a different type. This is where type casting comes in.
- The compiler automatically (implicitly) casts when the target type is wider than the source. There is usually no risk of information loss this way. However when we convert a long to a float for example, the least significant bits may be lost. However, the result of the conversion will be correctly rounded.
- When working with floats or longs, it is good practice to add f or L after a literal number. Such as `float temp = 1000.23f;` or `long temp = 1233L;`.
- We can also cast values explicitly. This is needed when the target is narrower than the source. This can be risky because the conversion may lead to information loss regarding magnitude as well as precision.
- The syntax for explicit casting is `(targetType) source`, for example `int temp = (int) doubleVariable;`
- In the above example, the fractional part of doubleVariable is lost and the rounded integer is stored in temp. Explicit casting may also lead to the value being truncated when the conversion would lead to a type overflow. For example, explicitly casting a huge long variable to an int variable.
- Similar to C++, we can increment a variable by typing `++var`, which increments a variable before using it. Whereas `var++` uses the variable before incrementing it.
- Note that the boolean type cannot be cast in Java, neither implicitly nor explicitly
- The syntax for comments in Java is identical to C++. However, there is an additional type of comment called ‘Java documentation comments’. It is used in conjunction with the javadoc tool.
- The Oracle Code Convention and Google Style Guide are the two main style guidelines for Java
- The Oracle Code Convention states that four spaces should be used as the unit of indentation throughout the program, and whitespace should be used inside statements needlessly.
- Java supports ternary operations just like other languages. The general syntax is `result = condition ? trueCase : elseCase;`
- The for loop works similarly to C++. All three parameters of the for loop are optional.
- In a while loop, the condition is checked each time **before** the code block is run, which makes it a “pre-test loop”
- On the other hand, Java also supports do-while loops which is a “post-test loop”, where the code blocks is executed first and the condition is tested afterwards. It’s syntax is

```
do {
    // body: do something
} while (condition);
```

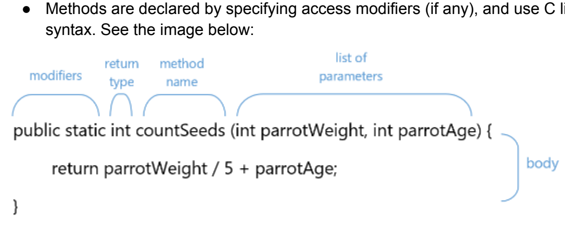
- Java supports `break` and `continue` statements in loop just as you’d expect
- In a Switch statement, the `break` keyword is optional. If a case is satisfied and it does not have a break keyword, the subsequent cases including the default case will be executed as well, until a break statement is encountered. For this reason, be careful about including the break statement. The following snippet illustrates this concept which is known as a “switch statement fallthrough”:

```
switch (digit)
{
  case 0:
    result = ZERO_DIGIT;
    break;

  case 1: //Switch statement fallthrough till case 9
  case 3:
  case 5:
  case 7:
  case 9:
     result = ODD_DIGIT;
     break;

  case 2:
  case 4:
  case 6:
  case 8:
     result = EVEN_DIGIT;
     break;
}
```

- Java supports “for-each” loops, where we can access each element of an array, string, or collection without having to use indexes. The following demonstrates the syntax:

```
for (type var : array) {
    //statements using var
}
```

- The key limitation of for-each loop is that the variable var will be a copy of the array element, and doesn’t refer to the array element itself. Therefore, we cannot modify the array in a for-each loop.
- Methods are declared by specifying access modifiers (if any), and use C like syntax. See the image below:



- The name of the method combined with the types of it’s parameters comprise its signature. In this example, the signature is countSeeds(int int)
- Here, we use the `public` “access modifier” to allow the method to be called from other classes. We use the `static` “non-access modifier” to tell JVM that this method can be called even if we haven’t instantiated an object of its class. It is called a “static method”
- An access modifier is basically a keyword that allows us to choose who can access a part of our code. Examples include `public`, `package-private`, `protected`, and `private`
- Because Java forces the programmer to use classes, the application is essentially a collection of objects that communicate by calling each other's methods. Even a simple procedural style program needs to have at least one class and its main method.
- The main method is typically declared as `public static` since it can be called from anywhere and we do not need to instantiate the class it is declared in.

Scanning the input

- Just like there’s a standard output, there’s also a standard input supported by the operating system. It is a stream of data that can be read into the program
- By default, it obtains data from the keyboard but it can also be redirected to a file
- The simplest way to read from the standard input is using the standard class `Scanner`, which can be imported by adding the following on top of your code: `import java.util.Scanner;`
- We declare a new Scanner object by calling its constructor as follows: `Scanner scanner = new Scanner(System.in);`
- Here, we pass `System.in` as the constructor parameter to indicate that we will be reading from standard input from the keyboard
- We can use the Scanner object’s `next()` method to read a single word from a string. This will always read one word as a string. If the input is numerical, it will still read it as a string rather than an integer
- The method `nextInt()` works similarly as next() but it returns an integer. `nextLong()` works in a similar manner.
- A similar method is `nextLine()`, which reads all data (including whitespace) until a newline is encountered
- A ‘token’ refers to a sequence of characters surrounded by whitespace. You can think of it as a ‘word’
- Methods `hasNext()` and `hasNextLine()` and `hasNextInt()` return a boolean indicating whether there is anything left to read. See the snippet below, which reads keeps reading and printing tokens until there is nothing left to read:

```
while(scanner.hasNext())
    System.out.println(scanner.next());
```

Object Oriented Programming

- Java provides strong support for object oriented programming.
- An object’s state is defined by the value of its fields and its behavior is defined by its methods. (A “field” or “attribute” is a synonym for a variable stored by the object)
- In OOP, an ‘interface’ is a class that does not contain any state, it only exists to be inherited from so it can provide an interface to its descendant classes.
- Mutability is a key concept in Java. The programmer can design mutable objects, where all of it’s fields are mutable. One can design weakly immutable objects, where some of its fields are immutable. Or strongly immutable objects, where all of its fields are immutable.
- Generally, objects of custom classes are mutable unless designed otherwise.
- OOP seeks to implement four principles: encapsulation which is where we combine data and operations into one single unit, abstraction which is where we hide the internal implementation from the programmer while only presenting relevant features, inheritance which is where allows for parent child relationships among classes where they share common logic, and polymorphism where we can have different implementations of the same method (works in conjunction with inheritance)
- Java supports method overloading. In case the exact method to be called is vague, the one with the ‘closest’ type to the argument is invoked in order of implicit casting. For example:

```
void method (int a) {...}
void method (long a) {...}
int temp = method(23);
//The first method is invoked, since literal integers are assumed to
be int by default.
```

- Java classes can have two types of methods, instance methods and static methods. Instance methods require the class to be instantiated (i.e. an object needs to be created) before the method can be called. Static methods can be called from the actual class itself even if it hasn’t been instantiated to create an object.
- Constructors have no return type (not even void), and have the same name as the class.
- Instance methods have access to the fields of the object. Static methods obviously do not since they can be called before instantiation (when the fields don’t exist!). The following snippet exemplifies:

```
class Human {
    String name;
    int age;

    public Human(String name, int age) {
        this.name = name;
        this.age = age;
    } //constructor (not used in this snippet)

    public static void averageWorking() {
        System.out.println("An average human works 40 hours per
week.");
    } //static method

    public void work() {
        System.out.println(this.name + " loves working!");
    } //instance method, accesses field using 'this' keyword
}

public static void main(String[] args) {

    Human.averageWorking(); // "An average human works 40 hours per
week."
    //averageWorking() works even though no object has been created
yet!
    Human peter =  new Human(); //instantiate class Human to create
object peter
    peter.name = "Peter";
    peter.work(); // //calling instance method on object peter
    Human alice =  new Human();
    alice.name = "Alice";
    alice.work(); // "Alice loves working!"
}
```

- If we do not explicitly define one of ourselves, Java automatically creates a “no argument constructor” that initializes all fields to their default value.
- We can have multiple constructors through overloading, and can even call one constructor from inside another constructor using `this()`
- When we call another constructor from inside a constructor, it must be the first line of the calling constructor’s code. The following snippet illustrates:

```
public class Robot {
    String name;
    String model;
    int lifetime;

    public Robot() {
        this.name = "Anonymous";
        this.model = "Unknown";
    }

    public Robot(String name, String model) {
        this(name, model, 20);
    }

    public Robot(String name, String model, int lifetime) {
        this.name = name;
        this.model = model;
        this.lifetime = lifetime;
    }
}
```

- Sometimes, objects may all share a field or method with the same value. Such fields, known as static members, may be declared with the `static` keyword.
- A static field is a class variable declared using the `static` keyword. It can store a primitive or a reference type. It has the same value for all objects of the class. Therefore, this field belongs to the class itself, not individual objects.
- If we want all instances of a class to have the same value for a field, it’s better to make this field static since it can save us memory.
- Static fields can be accessed from the class even if we haven’t instantiated any objects of the class, like so: `ClassName.fieldName;`
- Static fields can also be accessed from object instantiations of the class
- Here’s an interesting example where all instances of the class share a static field storing the current date. This date is updated every time a new object is instantiated:

```
public class SomeClass {

    public static Date lastCreated;

    public SomeClass() {
        lastCreated = new Date();
    }
}
```

- Note that static fields are not necessarily final or constant. They can be updated (either outside the class or by an instantiation)
- It is common for static fields, especially public static fields, to be constant. They are declared by using `static final` keywords in the variable declaration.
- By convention, final fields should be named in all caps and underscores
- Java also supports static methods. These methods can be called from the class name even if no object has been instantiated.
- For obvious reasons, static methods cannot access non-static fields (i.e. it doesn’t have access to `this` keyword). Also, static methods cannot call non-static methods. A static method can be called from the class without instantiation like `className.staticMethod(arg1, arg2)`, or from an object like `object1.staticMethod(arg1, arg2)`
- This is why the main method is static. It is called even though the class it is contained in isn't instantiated.
- In Java, an interface is a special type of class that cannot be instantiated. It is implemented by child classes who implement its methods. If a class implements an interface, it needs to implement all its declared abstract methods.
- An abstract method is a method with no body. It’s usually meant to be overridden by child classes.
- Note that “extends” and “implements” are two separate keywords in Java
- The interface only declares an abstract method, it doesn’t implement it. The implementation is done by its child classes. Here’s a snippet as an example:

```
interface DrawingTool {
    void draw(Curve curve);
}
class Pencil implements DrawingTool {
    ...
    public void draw(Curve curve) {...}
}

class Brush implements DrawingTool {
    ...
    public void draw(Curve curve) {...}
}
```

- This way, just a quick look at the classes Pencil and Brush inform us that it’s able to implement all the functionalities of DrawingTool. The purpose of interfaces is to declare functionality.
- Interfaces can be used as a type. See below:

```
DrawingTool pencil = new Pencil();
DrawingTool brush = new Brush();
```

- Now both pencil and brush are of the same type. Both of these objects can be treated similarly. This is a form of polymorphism. It allows us to write methods that can accept any of the interface implementations. See below, this function can accept either pencil or brush:

```
void drawCurve(DrawingTool tool, Curve curve)
```

- In an interface, all variables must be public static final. You don’t need to specify these access modifiers, the variables will be public static final by default.
- Methods are public abstract by default (these keywords are not required in an interface). An abstract method is one where it is not implemented, only the function signature is defined.
- You can implement default methods with implementation (default keyword required)
- You can implement static methods (static keyword required), and private methods (private keyword required). These must be implemented in the interface.
- Static methods can be invoked directly from the interface (i.e. even if we haven’t instantiated its child class).
- An interface cannot contain non-constant fields and it cannot have a constructor, since interfaces cannot be instantiated. See the snippet below:

```
interface Interface {
    int INT_CONSTANT = 0; //public static final by default
    void instanceMethod1();
    void instanceMethod2(); //These two functions are abstract
    static void staticMethod() { //static method
        System.out.println("Interface: static method");
    }
    default void defaultMethod() {
        System.out.println("Interface: default method. It can be
overridden");
    }
    private void privateMethod() {
        System.out.println("Interface private method");
    }
}

class Child implements Interface {
    //We use this annotation when we implement and override interface
methods
    @Override
    public void instanceMethod1() {
        System.out.println("Child: instance method1");
    }

    @Override
    public void instanceMethod2() {
        System.out.println("Child: instance method2");
    }
}

//Inside main method:
Interface temp = new Child();
temp.instanceMethod1(); //Child: instance method1
temp.defaultMethod(); //Interface: default method...
```

- An important feature in Java is that a class can implement multiple interfaces. Also, an interface can extend multiple interfaces using the `extends` keyword. See examples below:

```
interface A { }
interface B { }
interface C { }
//class D implements multiple interfaces
class D implements A, B, C { }

interface A { }
interface B { }
interface C { }
//We declare a new interface E which extends multiple interfaces
interface E extends A, B, C { }
```

- Furthermore, a class can both extend another class and implement an interface:

```
class A { }
interface B { }
interface C { }
class D extends A implements B, C { }
```

- Interfaces are useful because they allow polymorphism. Each class has its own unique implementation of the abstract class defined in the interface. However, the objects can have the reference type of their interface. This allows, among other things, for a method to call an object’s method without knowing the exact type of the object. For example:

```
void drawInstruments(DrawingTool[] instruments){
    for(instrument: instruments){
        instrument.draw();
    }
}
//in main method
DrawingTool pencil = new Pencil();
DrawingTool brush = new Brush();
```

- Here, the drawInstruments function has no idea whether it receives objects of type pencil or brush. It calls their draw method and each object is able to execute their own unique implementation of draw()
- An interface is closely related to an “abstract class”. An interface achieves 100% abstraction, while an abstract class allows partial abstraction. You can look up their differences online for more details.
- Sometimes, an interface may have no body at all. They are called “tagged interfaces” or “marker”. They are used to provide information to the JVM. A well known interface `Serializable` is an example.
- A top level class (not an inner class or a nested class) can have two access modifiers. `package-private` which is the default access modifier, or `public`
- In package-private, only other classes in the same package can access the class. For example, here PackagePrivateClass automatically has package-private access modifier even though we didn’t explicitly type it out:

```
package org.hyperskill.java.packages.theory.p1;
class PackagePrivateClass{
}
```

- Here, PackagePrivateClass will only be visible to other classes in `org.hyperskill.java.packages.theory.p1`
- Fields on the other hand can also have `private` and `protected` access modifiers. A common strategy is to make all fields private, and write getter and setter methods for those fields that need to be accessed or updated from outside the class. `package-private` is the default access modifier for fields too.
- The `protected` modifier makes it so that the field can be accessed by classes in the same package, as well as by its subclasses (including those in other packages)
- Inheritance refers to deriving a new class from a parent class, often acquiring some fields and methods from its parent. A class derived from another class is called a "subclass" or "child class" or "derived class"
- The class it is derived from is known as the "base class" or "superclass" or "parent class"
- We use the `extends` keyword when deriving a new subclass
- A key concept to note is that an object of type subclass is basically also an object of the supertype class. However, the reverse is not true.
- It's important to note that Java does not support multiple class inheritance. A subclass can only be derived from one superclass. (i.e. a class can only have one parent)
- However, more than one class can be derived from a class (i.e. a superclass can have multiple subclasses)
- Java supports multiple inheritance, so a subclass can be further derived into a sub-subclass.
- Subclasses inherit their parents' public and protected fields and methods, and also package-private fields and methods if the subclass is part of the same package as its parent. If a superclass would like to give access to its private variables to its child, a public or protected method (such as a getter or setter) can be implemented.
- It's important to note that constructors are not inherited. However, a subclass can invoke its parent's constructor using `super()`
- If you declare a class with the `final` keyword, no child classes can be derived from it. For example, `final class SuperClass { } //no child classes`
- The `super` keyword works similarly to `this`, except it refers to the superclass. For example, inside the constructor of a subclass you might have a statement like `super.val = val`. This sets the superclass's field.
- If we're going to invoke the superclass's constructor from the subclass constructor, the `super` keyword should be the first line in the constructor method. Here's an example:

```
class Person {
    protected String name;
    protected int yearOfBirth;
    protected String address;
    public Person(String name, int yearOfBirth, String address) {
        this.name = name;
        this.yearOfBirth = yearOfBirth;
        this.address = address;
    }
   Protected printInfo(){
        //Print name, yearOfBirth, and address
   }
}

class Employee extends Person {
    protected Date startDate;
    protected Long salary;
    public Employee(String name, int yearOfBirth, String address,
Date startDate, Long salary) {
        super(name, yearOfBirth, address); // invoking a constructor
of the superclass
        this.startDate = startDate;
        this.salary = salary;
    }
    Protected printInfo(){
        super.printInfo();
        //print startDate and salary
    }
}
```

- It's important to note that whenever a subclass's constructor is called, it calls its parent class's no-args constructor by default. This occurs even if we don't explicitly call the parent's constructor using `super()`. As the example shows, we can call `super()` explicitly in the child constructor with parameters if we'd need to.
- When we have superclasses and child classes, there are two ways to create a child object. We can use a subclass reference or a superclass reference. See the example below: (assume default no-args constructor is present)

```
class Person {
    protected String name;
    protected int yearOfBirth;
    protected String address;
}
//Assume getters and setters exist for these classes
class Client extends Person {
    protected String contractNumber;
    protected boolean gold;
}

class Employee extends Person {
    protected Date startDate;
    protected Long salary;
}

//inside main method
Client client = new Client(); //subclass reference
Person client2 = new Client(); //superclass reference
```

- As a general rule: If class A is a superclass of class B and class B is a superclass of class C then a variable of class A can reference any object derived from that class (for instance, objects of the class B and the class C). This is possible because each subclass object is an object of its superclass but not vice versa.
- Note that when we use a superclass reference to instantiate a subclass, we only have access to the fields and methods of the reference (the superclass) in this case. Continuing the example above:

```
client2.setName("Jennifer"); //this is okay client2 is of reference
superclass Person which has method a setName
client.setContractNumber("abc123"); //this is okay because client is
of reference subclass Client with a method setContractNumber
client2.setContractNumber("xyz321"); //not allowed! client2 is of
reference type Person which doesn't include this method
```

- We can always cast a subclass reference to its superclass. Then we can access members only present in the superclass. If we have an object of superclass reference but is an instance of a subclass, we can also cast it to a subclass reference. See the example below. We first cast a superclass reference (Person) to its subclass (Client). This is possible because the superclass reference (client2) is an instance of the subclass Client. In the next line, we cast a subclass (Client) to its superclass (Person), which is always allowed.

```
Client newClient = (Client) client2 //We have cast client2 to
subclass Client
Person newPerson = (Person) client //We have cast client to
superclass Person
```

- Two common use cases when we might want to use superclass references is when we have an array of objects of different types within the hierarchy (i.e a mix of superclass objects and subclass objects). Another common use case is when we have a method that accepts the superclass type but also works with subclasses. Here's an example to demonstrate:

```
public static void printNames(Person[] persons) {
    for (Person person : persons) {
        System.out.println(person.getName());
    }
}

Person person = new Employee();
person.setName("Ginger R. Lee");
Client client = new Client();
client.setName("Pauline E. Morgan");
Employee employee = new Employee();
employee.setName("Lawrence V. Jones");
Person[] persons = {person, client, employee};
printNames(persons);
```

- A subclass can have a method with the same function signature as one of its superclass's methods. This concept is known as method overriding (not the same thing as method overloading!)
- It allows the subclass to have its own unique implementation of a superclass method that's more specific. This is only possible when the subclass inherits a method from its superclass (i.e. you cannot override a superclass's private method, since the subclass doesn't inherit it). Here's an example:

```
class Mammal {
    public String sayHello() {
        return "ohlllalalalalalalaoaoaoa";
    }
}
class Cat extends Mammal {
    @Override
    public String sayHello() {
        return "meow";
    }
}
class Human extends Mammal {
    @Override
    public String sayHello() {
        return "hello";
    }
}
Mammal mammal = new Mammal();
System.out.println(mammal.sayHello()); // it prints
"ohlllalalalalalalaoaoaoa"
Cat cat = new Cat();
System.out.println(cat.sayHello()); // it prints "meow"
Human human = new Human();
System.out.println(human.sayHello()); // it prints "hello"
```

- Here's a code snippet that shows that Java correctly calls the subclass's override method even we use superclass references:

```
class Animal {
    public void say() {
        System.out.println("...An incomprehensible sound...");
    }
}

class Cat extends Animal {
    @Override
    public void say() {
        System.out.println("meow-meow");
    }
}

class Dog extends Animal {
    @Override
    public void say() {
        System.out.println("arf-arf");
    }
}

class Duck extends Animal {
    @Override
    public void say() {
        System.out.println("quack-quack");
    }
}

public class Main {
    public static void soundOff(Animal[] animals){
        for(Animal animal: animals){
            animal.say();
        }
    }

    public static void main(String[] args) {
        Animal duck = new Duck();
        Animal dog = new Dog();
        Animal cat = new Cat();
        soundOff(new Animal[] {duck,dog,cat});
       //quack-quack, arf-arf, and meow-meow are printed correctly!
    }
}
```

- You can invoke the base class method in the overridden method using the keyword `super`
- The overriding method should have the same or more lenient access modifier than the superclass's method.
- Static methods cannot be overridden
- Use the `final` keyword to prevent a method from being overridden. For example, `public final void method() {} //can't be overridden`
- If we have a method in a subclass with the same name as one in its superclass but with different parameters, they do not have the same signature. Therefore, this method will be unique to the subclass and does not override anything.
- If a superclass and subclass have static methods with the same function signature, the subclass's method will "hide" the superclass's version of this method
- A subclass and superclass are not allowed to have an instance method and a static method with the same signature. They can either both be static (in this case the superclass function is hidden), or they can both be instance methods (in this case overriding occurs)
- In Java there exists a root class named `Object` which is the default parent of all standard classes as well as custom classes. Every class extends the `Object` class implicitly.
- This class is in the java.lang package and is imported by default. Because it's the parent child of every class. Any object can be cast to the `Object` type. See the example below:

```
Object anObject = new Object();
Long number = 1_000_000L;
Object obj1 = number; // an instance of Long can be cast to Object
String str = "str";
Object obj2 = str; // the same with the instance of String
```

- The `Object` class provides methods that all subclasses (i.e. literally every class that exists) can access. It includes the following methods that can be handy in multithreaded programming: `wait`, `notify`, `notifyAll`
- It has the following methods useful for object identity, `hashCode` and `equals`
- It has the following methods for object management: `clone` and `getClass`. `clone` creates an identical copy of the object and returns it.
- It also contains a method called `toString()` which is used to return info on the object in human readable form. This method is often overloaded in our classes.
- `toString()` allows us to print the contents of an object right from System.out. For example, if human1 is the name of an object, we can print simply as `System.out.println(human1);` This is also handy when overriding classes that contain other classes as fields. However, be careful if two classes contain each other as their fields. This can cause an infinite recursion when we call `toString`, since the object will print its field, which will in turn print its field which is the original object we called toString on!
-

Strings

- It's important to note that strings are immutable in Java. It's value cannot be changed once it is declared.
- The benefit of immutable data is that they are thread safe: they can be shared by different threads safely
- Even though immutable objects cannot be changed, that doesn't mean the variable holding the immutable object cannot be reassigned. For example, see the snippet below:

```
String temp = "abc";
temp = temp + "def"; //this is allowed since we're assigning a new
value to variable temp instead of updating it's existing value
```

- We cannot use array indexes to access individual characters in a string. Furthermore, we cannot modify these characters since strings are immutable.
- We use String method charAt() whose parameter is an integer for the index of the character to be returned. In the example above, `temp.charAt(2)` returns 'c'.
- If we need to be able to modify individual characters, we can declare an array of characters (such as `char[] temp=new char[10];`) or use a `StringBuilder`.
- Characters in Java support Unicode (UTF-16) which is inclusive of ASCII. One can assign a unicode character by starting with the `\u`. For example, `char temp = \u0040;` sets temp to `'@'`
- You can also assign an integer to a character which represents a Unicode code. Characters can be operated on like they're integers. Java supports the usual gamut of escape sequences too.
- In Java, we compare two strings using the `compareTo` method. Alternatively, we can use the `equals` and `equalsIgnoreCase` methods
- Strings have a method called split(), which accepts a regex and limit as its parameters, and returns an array of strings surrounding the regex match. The following example illustrates:

```
String str = "geekss@for@geekss";
String[] arrOfStr = str.split("@", 5);
for (String a : arrOfStr)
    System.out.println(a); //prints geekss, for, and geekss
```

- The string object has numerous methods that can come in handy. You can look them up as necessary. One of these methods is `String.format()`, which allows you to create strings with other variables using printf-like syntax. For example, `String sf1=String.format("name is %s",name);`
- Strings can be concatenated both with other strings as well as different data types. When concatenating with other data types, they are automatically converted to the appropriate string value. See the snippet below:

```
String str = "str" + 10 + false; //str equals "str10false"
```

Advanced

- It is important to understand synchronous, asynchronous, and parallel processing. Synchronous is the basic case of doing things one at a time. Asynchronous processing is where parts of multiple tasks are done out of order, whether by a single or multiple executor. An example of asynchronous behaviour would be to continue with some other task while waiting for a fetch request to complete. Parallel processing is where multiple executors perform tasks individually and simultaneously.
- Every process must have at least one thread, and every thread must be part of a process. A process owns system resources and lends them to threads, schedules threads and facilitates inter thread communication.
- A process can be thought of as a self-contained unit of execution that has everything needed to accomplish its mission. It owns the resources and organizes the runtime environment.
- A thread is a stream of instructions from a process that can be scheduled and run independently. Each thread has its own executor, but multiple threads can be run in parallel if we have multiple executors. (By executor we're referring to something like a CPU core)
- A good analogy is that a process is a business, and threads are employees. The business owns the resources and allocates tasks, but it's threads who share the business's resources and actually do the work.
- Threads are useful because it is much more efficient to have threads share resources held by the thread, otherwise we'd need to rearrange access to resources every time a new process is created. With threads, we don't need to create new processes as often since we can just create new threads, which has a much lesser overhead than creating a new thread.
- If we have lightweight tasks, it is often better to timeshare these tasks in one thread instead of using multiple threads. This is known as "lightweight concurrency" or "internal concurrency". It is known as internal because it is contained within the thread
- Some methods are "instance methods", which can only be called once an object of that class has been created. See the example below. Here, `toLowerCase()` is an instance method. Note that it does not modify the `String` object. It returns a brand new `String` object (which can be reassigned to the String variable as shown in the example below)

```
String name = new String("Anya"); // created an instance (1)
name = name.toLowerCase(); // anya (2)
```

- In Java, all data types are either "primitive types" or "reference types". Primitive types are stored in the stack. In reference types, the actual data will be stored in the heap and a pointer to the heap address will be stored in the stack.
- Java has eight primitive types, which are lowercase. Reference types often begin with an uppercase letter
- In most cases, reference types are created with the `new` keyword, which allocates memory on the heap to store the object. This is called "instantiation", since we create an instance of the class. See the example below:

```
String language = new String("java");
String language2 = "java";
```

- In this example, the first line is the typical of instantiating a reference object. The second method is String specific and equivalent to the first line.
- If you execute the new keyword twice on the same variable, a brand new object is instantiated and assigned to the variable. The object that was previously attached to the variable is lost unless its reference was assigned to another variable before the reassignment. See the snippet below to demonstrate:

```
String temp = new String("Java");
temp = new String("Javascript"); //A new object is instantiated and
it's reference is stored in temp. The previous object containing
"Java" is now lost and will be picked up by the garbage collector
```

- The way assignment works is significantly different in reference and primitive types. In primitive types, the actual value of the primitive is *copied*, so the new variable is an independent copy from the first. With reference types, only the address to the heap memory is copied. This means that with heap assignment, the new copy is literally the same data. See the example below to demonstrate:

```
Int temp = 2020;
Int temp2 = temp2; //int is a primitive type, so the value 2020 is
copied
//temp and temp2 are completely independent variables
String temp3 = "Example";
String temp4 = temp3; //Only a reference to the String containing
"Example" is copied
//temp3 and temp4 are not really independent, one can influence the
other
```

- That being said, because the `String` type is immutable, if we were to assign new text to temp3 in the example above, temp4 would continue to hold "Example" because a new address would be initialized when we reassign temp3.
- Because Strings are immutable, they behave much like primitive types in practice.
- Whenever we reassign a String variable, a brand new object is instantiated and its address is stored in the variable. The String object this variable held before is either lost, or is held by any other variable it was assigned to.
- Because of the way reference types work, comparisons using == and != don't work the way they do for primitive types. The comparison operator compares the reference addresses, not the contents of the data itself. The following example illustrates:

```
String s1 = new String("java");
String s2 = new String("java"); //This string is different and has a
different address from s1
String s3 = s2; //Only the address is copied, not the actual data in
the heap
System.out.println(s1 == s2); // false because s1 and s2 point to
different addresses
System.out.println(s2 == s3); // true because they have the same
address
```

- With reference types, we can set it to null which can indicate that it hasn't been instantiated yet or doesn't have a value. This does not work with primitive types. For example, `String s1 = null;`
- In Java, arrays are a reference type and thus need to be instantiated using the `new` keyword. Arrays hold a fixed size of the same datatype sequentially (the size cannot be changed once the array has been instantiated)
- An array data type can be declared as follows: `int[] array;` but this does not allocate any memory since we haven't used the `new` keyword yet.
- We can declare and instantiate an array as follows: `int[] numbers = new int[n];`
- Since we haven't enumerated the contents of the array, they are initialized to the default value of their datatype (0 in the case of `int`)
- Alternatively, the following syntax also instantiates an array even though we haven't used the `new` keyword: `int[] numbers = { 1, 2, 3, 4 };`
- The array object has a field called length, which returns the capacity of the array. For example, `numbers.length //returns 4`
- We have access to some handy array methods if we import utility class `Arrays`. We can import it using `import java.util.Arrays;`
- The following snippet demonstrates some of these functionalities:

```
int[] famousNumbers = { 0, 1, 2, 4, 8, 16, 32, 64 };
Arrays.sort(famousNumbers);
String arrayAsString = Arrays.toString(famousNumbers);
Arrays.equals(numbers1, numbers2); //returns true if numbers1 and
numbers2 are arrays of the same length containing the same value at
each index
Int size = 10;
char[] characters = new char[size];
Arrays.fill(characters, 0, size / 2, 'A'); //fill array starting at
index 0, up to and excluding size/2 with 'A'
```

- Because arrays are a reference type, when you pass an array to a method, any changes you make to the array inside the method will reflect outside the method as well.
- We can accept an unknown number of parameters in a method using varargs (variable length arguments). We declare a vararg by typing three dots before the type.
- When a method accepts two or more parameters, the varargs must be the last one in the function signature. See the snippet below, where we declare a vararg named varPam

```
public static void method(int a, double... varPam) { /* do something
*/ }
method(1, 2, 3); //a is 1, and varPam is [2,3]
method(1, new int[] { }); // a is 1, no arguments here for varPam
```

- As you can see, varargs can accept either separate integers in the method call (separated by commas), or it can accept an array of integers.
- Java lets us declare `final` variables which cannot be modified once assigned. These variables are called constants. It is standard practice to name constants using all capital letters and underscores. See the snippet below to demonstrate some concepts:

```
final int temp; //this is allowed, don't need to assign at the same
time as declaration
System.out.print(temp); //this causes an error, we cannot use a
constant before assigning it
temp = 30; //we now assign this constant. It cannot be modified going
forward
temp = temp + 1; //error, we cannot modify a constant
```

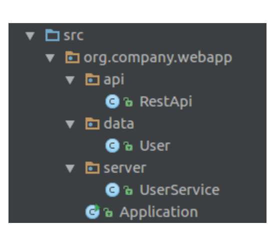
- When we use the final keyword with a reference type, it only prevents us from reassigning the constant variable. It does not prevent us from changing the internal state of the object. This is because at its core a reference variable only stores an address to heap memory. With a final reference type, you cannot change the address stored in the constant (via reassignment), but we can change the data in the heap location it points to (changing the internal contents of the object)
- Note that the `final` keyword is used in other contexts in Java in addition to declaring constant variables (more research may be needed)
- Java has support for "annotations", which provide metadata and mark classes, variables, methods, etc.
- Annotations can be used to provide information to the compiler, to provide info to the IDE (to generate code, etc), or to provide info to frameworks and libraries at runtime
- The syntax for including an annotation is the `@` symbol followed by the name of the annotation. Java has three built-in annotations, `@Deprecated` which indicates that the marked element (class, method, etc) is deprecated and should be avoided.
- `@SuppressWarnings` tells the compiler to disable compile-time warnings. `@SuppressWarnings` must be specified with annotation parameters to tell the compiler what type of warnings to ignore.
- `@Override` is used to mark a method that overrides a superclass method. This annotation can only be applied to methods.
- Some annotations accept elements. These elements have a name and type. For example, `@SuppressWarnings` accepts an element called "value". This annotation has no default value, so the value element must always be specified. See the following snippet:

```
@SuppressWarnings(value = "unused") //Suppress warnings about unused
variables
@SuppressWarnings("unused") //This is allowed when we have just one
element named "value"
@SuppressWarnings({"unused", "deprecation"}) //Passing array as value
```

- Custom annotations make take the following form:

```
@Range(min = 1, max = 100)
private int level = 1; //level must store an integer between 1 and
100
@NotNull
public String getLogin() {
    return login; //getLogin must not return a null value
}
Public void doSomething (@Range(min = 1, max = 100) int level){
     //Annotations can be used in method parameters like this
}
```

- Java applications typically consist of numerous classes and they can be difficult to manage if they're stored in the same directory. This is where "packages" come in.
- A package provides a mechanism to store related classes in a module (package)
- A package can contain other packages, and the whole structure resembles directories in a file system.
- It allows us to group related classes, which makes it easier to figure out where a class is. It also helps avoid class name conflicts. It also helps control access with access modifiers
- It is standard convention that package names must always be in all lower case letters. Take a look at the example below:



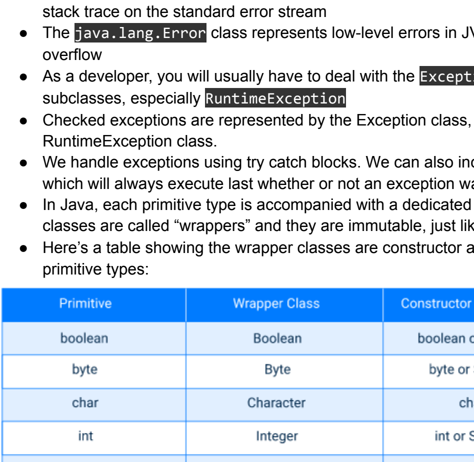
- Here, the full name of the User class is actually org.company.webapp.data.User
- We can print the full name of this class as follows: `System.out.println(User.class.getName()); // org.company.webapp.data.User`
- Classes declared as part of a package have a keyword `package` on top of the file. For example,

```
package org.company.webapp.data;
public class User {
}
```

- Packages help avoid conflicts in class names. Since even if two classes have the same name, their full name (including the package) will be different. (Unless of course there is also a conflict in the package name!)
- To avoid package name conflicts, it is good practice to begin the name of your package with your domain name, in reverse. For example, org.hyperskill
- We use the `import` keyword to be able to use classes defined in another package
- If we're using both the `package` and `import` keywords in a file, the package must come first.
- We can use a `*` to import all classes from a package, but you should avoid doing this if you can help it. For example `import java.util.*;`
- We can use classes from other packages without using the `import` keyword if we write out its full name, for example `java.util.Scanner scanner = new java.util.Scanner(System.in);`
- We can import static fields and methods of a class by using the static keyword in our import statement. Furthermore, if we use a `*` in this statement, we don't need to mention the class name before invoking its static methods. For example, here we import all static methods from class `Arrays`:

```
import static java.util.Arrays.*;
//in main method, we have an array called numbers
sort(numbers); // instead of writing Arrays.sort(...)
```

- If we do not write a package statement before defining a class, it will be placed inside the "default package". It has a big disadvantage: classes inside the default package cannot be imported into classes inside named packages. For any "real world" use, you ought to define package names.
- In Java, exceptions are objects of classes that exist in a hierarchy. See the diagram below:



- As you can see, the base class for all exceptions is `java.lang.Throwable`
- Its two direct subclasses are `java.lang.Error` and `java.lang.Exception`
- We have access to the following methods, `getMessage` which returns a String with details about this exception object, `getCause` which returns a Throwable object with the cause of this exception, and `printStackTrace` which prints a stack trace on the standard error stream
- The `java.lang.Error` class represents low-level errors in JVM like stack overflow
- As a developer, you will usually have to deal with the `Exception` class and its subclasses, especially `RuntimeException`
- Checked exceptions are represented by the Exception class, excluding the RuntimeException class.
- We handle exceptions using try catch blocks. We can also include a finally block, which will always execute last whether or not an exception was encountered.
- In Java, each primitive type is accompanied with a dedicated class. These classes are called "wrappers" and they are immutable, just like `String`.
- Here's a table showing the wrapper classes are constructor arguments for all the primitive types:

Primitive	Wrapper Class	Constructor Argument
boolean	Boolean	boolean or String
byte	Byte	byte or String
char	Character	char
int	Integer	int or String
float	Float	float, double or String
double	Double	double or String
long	Long	long or String
short	Short	short or String

- The term "boxing" refers to converting a primitive type to an object of its wrapper class. See the example below:

```
int primitive = 100;
Integer reference = Integer.valueOf(primitive); // boxing
int anotherPrimitive = reference.intValue(); //unboxing
```

- As you can see `valueOf` is a static method of the Integer class that returns an object of its wrapper type. `intValue()` is an instance method that returns the value of this wrapper object.
- "Autoboxing" and "auto-unboxing" is simpler syntax for doing the same thing. They are automatic conversions handled by the compiler See the example below:

```
double primitiveDouble = 10.8;
Double wrapperDouble = primitiveDouble; // autoboxing
double anotherPrimitiveDouble = wrapperDouble; // auto-unboxing
```

- Note that autoboxing only works when the left and right side of the assignment are of the same type.
- When we want to create wrapper objects using other types, we can use their constructor methods. See the example below:

```
Integer number = new Integer("10012"); // an Integer from "10012"
Float f = new Float("0.01");           // a Float from "0.01"
Long longNumber = new Long("100000000"); // a Long from "100000000"
Boolean boolVal = new Boolean("true");
```

- You can also use static methods "special methods" as follows:

```
Long longVal = Long.parseLong("1000");      // a Long from "1000"
Long anotherLongVal = Long.valueOf("2000"); // a Long from "2000"
```

- Here, if we pass a non-numerical String as the parameter, we will get a `NumberFormatException`
- Wrapper constructors have been deprecated, so you should use special methods instead.
- Because wrappers are a reference type the == operator only compares their addresses, not the content they hold. Use the equals method for this purpose. See the example below:

```
Long i1 = Long.valueOf("2000");
Long i2 = Long.valueOf("2000");
System.out.println(i1 == i2);      // false
System.out.println(i1.equals(i2)); // true
```

- Beware that unboxing a wrapper object can cause a null pointer error if the object is null. To prevent this, we can check to make sure the object is not null before trying to unbox it.
- You can perform arithmetic operations on wrapper objects just like you would with primitive types.
- A key reason to use wrapper objects is that they can be used in standard collections (like list, set), while primitives cannot.
-

Threads

- Java was designed with multithreaded programming in mind. Thread functionality is contained in `java.lang.Thread`. Every Java program has at least one thread called main, created automatically by JVM to execute statements inside the main function.
- Java applications have some other threads by default (for example, there is a thread for garbage collector)
- Each thread is represented as an object, an instance of the `java.lang.Thread` class. It has a static method called `currentThread` which returns a reference object to the thread that's currently being executed. For example, `Thread thread = Thread.currentThread();`
- Each thread object has a name, an identifier of type long, a priority, and other characteristics. The object has methods to get these attributes. There is also a setter method to change its name. See the example below:

```
public class MainThreadDemo {
    public static void main(String[] args) {
        Thread t = Thread.currentThread(); // main thread

        System.out.println("Name: " + t.getName());
        System.out.println("ID: " + t.getId());
        System.out.println("Alive: " + t.isAlive());
        System.out.println("Priority: " + t.getPriority());
        System.out.println("Daemon: " + t.isDaemon());

        t.setName("my-thread");
        System.out.println("New name: " + t.getName());
    }
}
```

- `isAlive()` returns a boolean indicating whether the thread has been started and hasn't died yet.
- Threads with a higher priority are executed with higher preference.
- The main thread is our starting ground from where we can spawn new threads to perform tasks.
- There are two ways to create our own thread: we can write our own class that extends the `Thread` class and overwrites its `run` method:

```
class HelloThread extends Thread {

    @Override
    public void run() {
        String helloMsg = String.format("Hello, i'm %s", getName());
        System.out.println(helloMsg);
    }
}
//inside main
Thread t1 = new HelloThread(); // a subclass of Thread
```

- Or, we can implement an already existing interface called `Runnable`, and pass its implementation to the constructor of Thread:

```
class HelloRunnable implements Runnable {
    @Override
    public void run() {
        String threadName = Thread.currentThread().getName();
        String helloMsg = String.format("Hello, i'm %s", threadName);
        System.out.println(helloMsg);
    }
}
//inside main
Thread t2 = new Thread(new HelloRunnable()); // passing runnable
```

- Either way, we need to override the `Thread` class's `run()` method
- You can specify a name for your thread by passing to the Thread constructor as follows: `Thread myThread = new Thread(new HelloRunnable(), "my-thread");`
- We can pass whatever we want in our constructor, such as an array for example. Changes made by the thread to the array would be reflected in other threads as well.
- The Thread class has an instance method called `start()` which is used to start the thread you've created. It actually creates a new thread and executes the contents of its run function. However, the thread's run() method will not start executing immediately, there will be a small delay.
- By default, threads run in "non-daemon" mode. In non-daemon mode, JVM will not terminate the program while the non-daemon thread is running. On the other hand, a daemon thread does not prevent JVM from terminating the program.
- The code inside a thread is executed sequentially. However, we cannot determine the relative order of statements among different threads, including the main thread, without explicit measures. This is especially because we do not know how long after we call `start()` for a thread that it's `run()` method will actually start executing.
- Basically, we cannot rely on the order of execution between multiple threads unless special measures have been taken.
- Here's an example of a thread that reads integers from standard input and prints their square. It keeps doing this until the user inputs 0:

```
class SquareWorkerThread extends Thread {
    private final Scanner scanner = new Scanner(System.in);

    public SquareWorkerThread(String name) {
        super(name);
    }

    @Override
    public void run() {
        while (true) {
            int number = scanner.nextInt();
            if (number == 0) {
                break;
            }
            System.out.println(number * number);
        }
        System.out.println(String.format("%s finished", getName()));
    }
}
```

- Our class that extends `Thread`, or our class that implements runnable, may have private variables and a constructor to handle them accordingly. For example,

```
Runnable task = new PrintMessageTask("Hi, I'm good.");
Thread worker = new Thread(task);
```

- Here, we have a class called PrintMessageTask that implements `Runnable`. It's constructor includes reading a `String` parameter and initializing a private `String` variable to it. We then create a new `Thread` with this runnable.
- We also have the ability to manipulate threads while they are running. Two common methods to achieve this are `sleep()` and `join()`. Both of these throw a checked `interruptedException`
- The static method `Thread.sleep()` suspends execution of the current thread for a specified number of milliseconds.
- Another way to suspend a Thread is to use class `TimeUnit` from `java.util.concurrent`. `TimeUnit.MILLISECONDS.sleep(2000)` calls `Thread.sleep()` for 2000 milliseconds, while `TimeUnit.SECONDS.sleep(2)` also suspends execution for two seconds
- The join() method makes the current thread wait until this other thread is finished. See the example below to demonstrate:

```
//We have a class Worker that extends the Thread class
Thread worker = new Worker();
worker.start(); // start the worker
System.out.println("Do something useful");
worker.join(3000L);  // waiting for the worker
System.out.println("The program stopped"); //This will print after
worker is done
```

- Here, we start a new thread called worker, and make the current thread wait until it's finished until we go on to print the final statement. Notice that we can pass a float parameter, which will limit the amount of time the current thread will wait in milliseconds. In this example, the current thread will wait until worker is done, or it'll resume after three seconds if the worker still isn't done.
- If an error occurs in a thread that is not caught and handled by a method, the thread will be terminated. If we are running a single threaded program, this means the whole program will end. This is because JVM terminates a program as soon as there are no more non-daemon threads active.
- This is the case even when the main thread has an error. The program will continue until other threads finish.
- Keep in mind that even if a thread ended with an error, the other threads will continue on. If thread A is waiting for thread B to finish using `join()`, and thread B has an exception, thread A will resume its execution form its next statement after `join()`
- Exceptions in threads are handled independently. It is good practice to write exception handlers especially when dealing with multi-threaded programs.
- All threads in a process share the same heap memory. This can be used to facilitate inter thread communication.
- If multiple threads are working on the same data concurrently, there are a few points we ought to keep in mind:
- Not all operations are atomic. A non-atomic operation is an operation consisting of multiple steps. If a thread operates on an intermediate value of a non-atomic operation being performed by another thread, we run into a problem called "thread interference". It is where the sequence of steps for a non-atomic operations may overlap between threads.
- Changes of a variable performed by one thread may be invisible to others
- If changes are visible, their order might not be (reordering)
- See the snippet below to demonstrate:

```
class Counter {
    private int value = 0;
    public void increment() {
        value++;
    }
    public int getValue() {
        return value;
    }
}

class MyThread extends Thread {
    private final Counter counter;
    public MyThread(Counter counter) {
        this.counter = counter;
    }
    @Override
    public void run() {
        counter.increment();
    }
}

//Inside main method
Counter counter = new Counter(); //default no-args constructor, value
is zero
MyThread thread1 = new MyThread(counter);
MyThread thread2 = new MyThread(counter);
//We create two threads with the same counter!
thread1.start(); // start the first thread
thread1.join();  // wait for the first thread
thread2.start(); // start the second thread
thread2.join();  // wait for the second thread
System.out.println(counter.getValue()); // it prints 2
```

- Here, we don't have to deal with any issues since the main method waits for thread1 to finish before starting thread2.
- Even though Counter's method increment() only has one line of code, value++, it is still not an atomic operation. It in fact has three distinct operations: read the value, increment it, and write it back to the value variable.
- For this reason, it is prone to thread interference if two threads call increment() on the same instance of counter around the same time.
- For example, thread A may read the value zero. It then increments it by one (but it hasn't written this new value back to the variable yet). Then, thread B reads the value (still zero) and increments it by one. Now, thread A gets around to writing the updated value to the variable (one). Finally, thread B writes the updated value it calculated to the variable (also one!). Therefore, even though increment was called twice, because of thread interference the value of the counter only increased by one.
- Note that the reading and writing of primitive types (except `long` and `double`) are guaranteed to be atomic. `long` and `double` types can also be made atomic by declaring them with the `volatile` keyword.
- For various reasons such as compiler optimization, caching, etc, a change in a value by one thread may not be visible to another thread. We can avoid this problem by declaring the variable with the `volatile` keyword. This keyword may be used in an instance variable or a static variable.
- Note that the `volatile` keyword still doesn't make increment and decrement operations atomic.
- Java provides ways to synchronize threads to avoid some of these issues.
- A "critical section" is code that accesses shared resources and should not be executed by more than one thread at the same time. This resource could be a variable, an external file, a database, or anything else. In the previous example, the increment operation in the Counter is a critical section.
- A "monitor" is a mechanism in Java to control access to objects. Each class and object has an implicit monitor. If a thread acquires a monitor for an object, then other threads must wait until the owner (the thread that has the monitor) releases it. Then the other thread can take ownership of the object by acquiring its monitor.
- Thus, a thread can be locked out from an object through its monitor, until the monitor is released. This allows programmers to protect critical sections from being executed at the same time by different threads.
- The "classic" and simplest way to protect critical sections is by using the keyword `synchronized`. We may have synchronized methods (either static or instance method), or synchronized blocks or statements (inside a static or instance method).
- When we declare a method or block as synchronized, the monitor for the object it is in ensures that only one thread can access the synchronized method or block (the critical section) at a time.
- When we declare a static method as synchronized, the monitor is the class itself. Only one thread can execute the body of this method at a time.
- If an instance method is declared as synchronized, the monitor is the instance (the object). In this case, only one thread can execute this particular object's method at a time. It does not stop threads from executing this method on other objects at the same time. For example, if we have two objects of the same type called A and B, and they have a method declared as `public synchronized void doSomething()`, two threads can execute doSomething on A and B simultaneously, since it is an instance method and A and B have their own monitors. However, two threads cannot execute doSomething on A simultaneously.
- You may also create a synchronized block, when a part of the body of a method is the critical section. See the example below:

```
class SomeClass {
    public static void staticMethod() {
        // unsynchronized code ...
        synchronized (SomeClass.class) { // synchronization on the
class
            // synchronized code
        }
    }
    public void instanceMethod() {
        // unsynchronized code
        synchronized (this) { // synchronization on this instance
            // synchronized code
        }
    }
}
```

- As you can see, we need to specify the object that holds the monitor and can lock the thread. In the case of a static method, it is the class itself. In an instance method, it is the object (specified with the `this` keyword)
- Any changes made by a thread inside a synchronized method or block will be visible to other threads after they acquire the monitor. This is one of the reasons why it might be a good idea to synchronize getter methods as well, since if the getter is not synchronized, a thread might need to acquire any monitor if it's only calling the getter. Therefore, it is not guaranteed that this thread will get the latest value.
- Going back to the Counter example, we can avoid thread interference simply by declaring the increment and getValue methods as synchronized. The getValue method should be synchronized to ensure that we return the correct updated value after it has been incremented.
- It is not necessary to synchronize methods that only read shared data (such as by calling a getter method) if we only read after writer threads have finished execution. We ensure that reading takes place after writer threads are done using `join()`. This is the case in the Counter example above, where only the main thread calls `getValue()` after the writer threads have finished. Therefore, this example would work even if getValue wasn't synchronized (increment still needs to be synchronized, however).
- It is also not necessary to synchronize methods that only read shared data (such as by calling a getter method), if the resource in question is a `volatile` variable.
- It's important to note that when a class has multiple synchronized methods or blocks, they all share the same monitor which is anchored to the object. Therefore, when one synchronized instance method is being executed, other threads cannot execute the other synchronized methods or blocks either. See the snippet below:

```
class SomeClass {
    public synchronized void method1() {
        // do something useful
    }
    public synchronized void method2() {
        // do something useful
    }
    public void method3() {
        synchronized (this) {
            // do something useful
        }
    }
}
```

- Here, all both methods as well as the block are synchronized using the same monitor (the object itself). Therefore, only one thread can execute one of these methods at a time per object. It is not possible, for example, for one thread to execute method1 and another thread to execute method2 at the same time on the same instance. Both methods and the block share the same monitor, so only one of them can be executed at a time regardless of how many threads have access to the object.
- Similar behavior as the above exists when we have multiple synchronized static methods.
- A thread cannot acquire a lock held by another thread, but it can acquire a lock it already owns, this is called "reentrant synchronization". See the snippet below:

```
class SomeClass {
    public static synchronized void method1() {
        method2(); // legal invocation because a thread has acquired
monitor of SomeClass
    }
    public static synchronized void method2() {
        // do something useful
    }
}
```

- If we need multiple locks per object, we can instantiate new objects and use their locks. This is useful when we have multiple fields that are independent, and can thus safely be updated at the same time by different threads on the same object. This technique is called "fine grained synchronization". If we lock both these fields using the same monitor, then they cannot be updated at the same by two threads, thus reducing performance. See the example below to demonstrate:

```
class SomeClass {
    private int numberOfCallingMethod1 = 0;
    private int numberOfCallingMethod2 = 0;
    final Object lock1 = new Object(); // an object for locking
    final Object lock2 = new Object(); // another object for locking
    public void method1() {
        System.out.println("method1...");
        synchronized (lock1) {
            numberOfCallingMethod1++;
        }
    }
    public void method2() {
        System.out.println("method2...");
        synchronized (lock2) {
            numberOfCallingMethod2++;
        }
    }
}
```

- Here, we have two fields to count the number of times method1 and method2 are called. Because these fields are totally independent, two threads can safely call method1 and method2 at the same time on the same instance. We allow this by synchronizing method1 and method2 using separate monitors. We create two objects, lock1 and lock2, for this purpose.
- Because synchronization reduces parallelism and performance, minimize synchronization when possible, synchronize blocks instead of whole methods when appropriate, and use fine grained synchronization when appropriate instead of using a single lock.