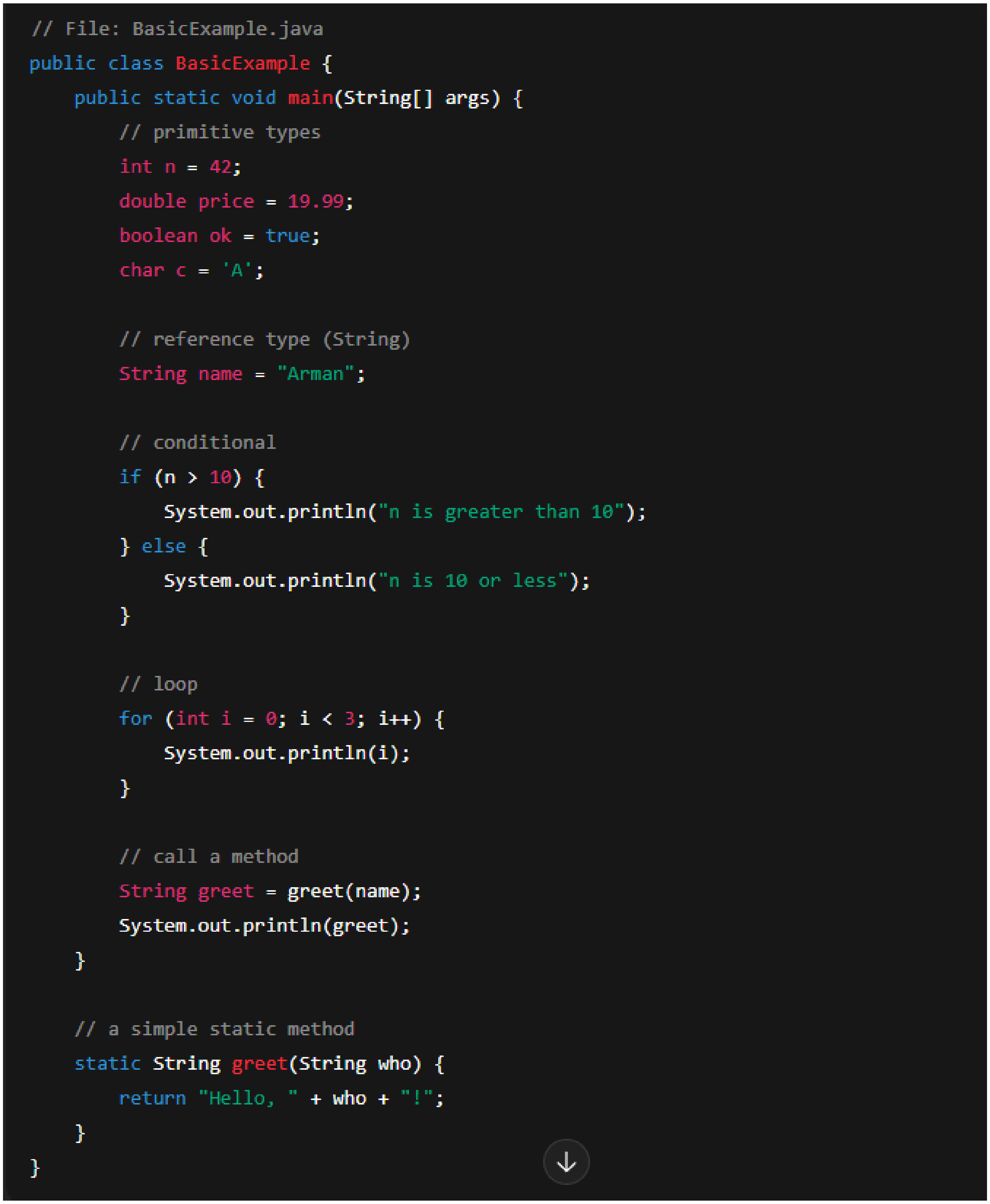
JAVA

Basic — fundamentals & simple programs

1) Hello world + basic types, control flow, methods, classes



```java
// File: BasicExample.java
public class BasicExample {
    public static void main(String[] args) {
        // primitive types
        int n = 42;
        double price = 19.99;
        boolean ok = true;
        char c = 'A';

        // reference type (String)
        String name = "Arman";

        // conditional
        if (n > 10) {
            System.out.println("n is greater than 10");
        } else {
            System.out.println("n is 10 or less");
        }

        // loop
        for (int i = 0; i < 3; i++) {
            System.out.println(i);
        }

        // call a method
        String greet = greet(name);
        System.out.println(greet);
    }

    // a simple static method
    static String greet(String who) {
        return "Hello, " + who + "!";
    }
}
```

Explanation (basic)

- public class BasicExample — declares a class; Java source files typically contain one public class with the same filename.
- public static void main(String[] args) — program entry point.
- Primitive types: int, double, boolean, char.
- String is a reference type (immutable).
- if / else and for are standard control structures.
- greet is a static method — can be called from main without an instance.

2) Arrays and simple collections

```java
import java.util.Arrays;
import java.util.List;
import java.util.ArrayList;

public class ArrayExample {
    public static void main(String[] args) {
        int[] nums = {1,2,3};
        System.out.println(Arrays.toString(nums));

        List<String> list = new ArrayList<>();
        list.add("apple");
        list.add("banana");
        for (String s : list) {
            System.out.println(s);
        }
    }
}
```

- int[] is a primitive array.
- List is an interface; ArrayList is a concrete implementation.
- for-each loop (for (T item : collection)).

Intermediate — OOP, exceptions, generics, streams, I/O

1) Classes, inheritance, interfaces, overriding

```java
// File: AnimalDemo.java
interface Speak {
    String sound();
}

class Animal {
    protected String name;
    public Animal(String name) { this.name = name; }
    public String describe() { return "Animal: " + name; }
}

class Dog extends Animal implements Speak {
    public Dog(String name) { super(name); }
    @Override
    public String sound() { return "Woof"; }

    // method overriding
    @Override
    public String describe() {
        return "Dog: " + name + " says " + sound();
    }
}

public class AnimalDemo {
    public static void main(String[] args) {
        Animal a = new Dog("Rex");
        System.out.println(a.describe()); // polymorphism: Dog.describe()
        // cast to access Dog-specific or interface methods
        Speak s = (Speak) a;
        System.out.println(s.sound());
    }
}
```

- interface defines a contract (methods without implementations, before Java 8).
- extends for inheritance, implements for interfaces.
- @Override documents method overriding.
- Polymorphism: a Dog referenced as Animal calls overridden method at runtime.

2) Exceptions, try-with-resources, and simple I/O

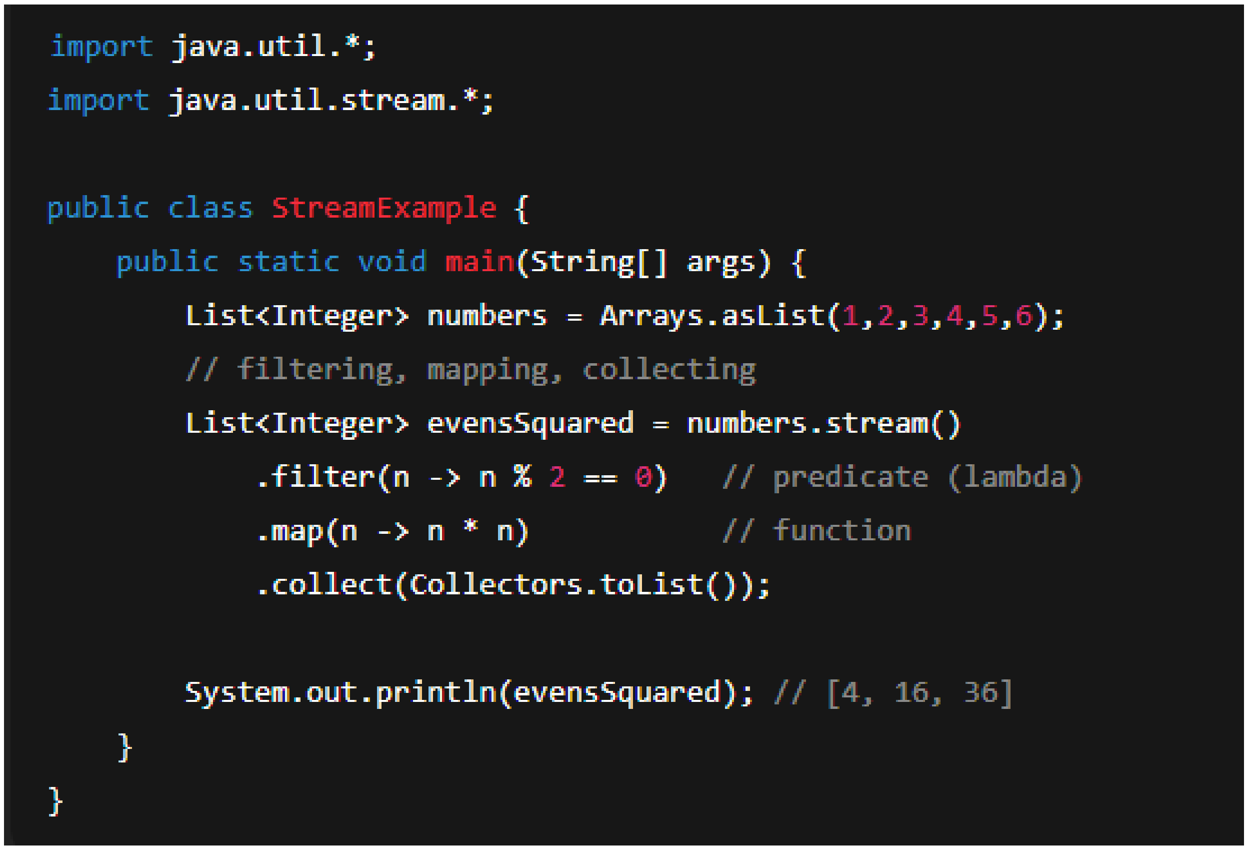
```java
import java.io.*;

public class IOExample {
    public static void main(String[] args) {
        String path = "example.txt";
        try (BufferedWriter w = new BufferedWriter(new FileWriter(path))) {
            w.write("Hello file");
        } catch (IOException e) {
            e.printStackTrace();
        }

        try (BufferedReader r = new BufferedReader(new FileReader(path))) {
            String line;
            while ((line = r.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- try-with-resources ensures AutoCloseable resources are closed.
- IOException is a checked exception — must be handled or declared.

3) Generics + Collections + Streams (Java 8+)



```java
import java.util.*;
import java.util.stream.*;

public class StreamExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1,2,3,4,5,6);
        // filtering, mapping, collecting
        List<Integer> evensSquared = numbers.stream()
            .filter(n -> n % 2 == 0)   // predicate (lambda)
            .map(n -> n * n)           // function
            .collect(Collectors.toList());

        System.out.println(evensSquared); // [4, 16, 36]
    }
}
```

- **Generics: List<Integer> ensures type safety at compile time.**
- **Streams support functional-style operations: filter, map, collect.**
- **Lambdas: n -> n % 2 == 0.**

Advanced — concurrency, JVM, reflection, modules, advanced patterns

1) Concurrency: Threads, Executors, synchronization, CompletableFuture

```java
import java.util.concurrent.*;

public class ConcurrencyExample {
    public static void main(String[] args) throws Exception {
        ExecutorService pool = Executors.newFixedThreadPool(4);

        Callable<Integer> task = () -> {
            // simulate work
            Thread.sleep(500);
            return 7;
        };

        Future<Integer> future = pool.submit(task);

        // do other work...
        System.out.println("Doing main thread work...");

        // get result (blocking)
        Integer result = future.get(); // InterruptedException, ExecutionException
        System.out.println("Result = " + result);

        // CompletableFuture example (non-blocking composition)
        CompletableFuture<Integer> cf = CompletableFuture.supplyAsync(() -> {
            return 21;
        }, pool).thenApply(x -> x * 2);

        System.out.println("CF result: " + cf.get());

        pool.shutdown();
    }
}
```

- Use ExecutorService instead of manually creating Threads.
- Future.get() blocks; CompletableFuture enables async composition.
- Handle InterruptedException and ExecutionException.

2) Concurrent collections & locks

```
import java.util.concurrent.*;
import java.util.*;

public class ConcurrentCollections {
    public static void main(String[] args) {
        ConcurrentMap<String, Integer> cmap = new ConcurrentHashMap<>();
        cmap.put("a", 1);
        cmap.compute("a", (k, v) -> v == null ? 1 : v + 1); // atomic compute

        BlockingQueue<String> q = new ArrayBlockingQueue<>(10);
        q.offer("x"); // non-blocking offer
    }
}
```

- **ConcurrentHashMap avoids global locking, good for high concurrency.**
- **BlockingQueue useful for producer-consumer patterns.**

3) Reflection, annotations, and custom annotation

```java
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface RunMe { }

class ReflectionDemo {
    @RunMe
    public void hello() {
        System.out.println("hello via reflection");
    }
}

public class ReflectionMain {
    public static void main(String[] args) throws Exception {
        ReflectionDemo obj = new ReflectionDemo();
        for (Method m : ReflectionDemo.class.getDeclaredMethods()) {
            if (m.isAnnotationPresent(RunMe.class)) {
                m.invoke(obj); // call annotated method
            }
        }
    }
}
```

- Annotations can be retained at runtime (RetentionPolicy.RUNTIME).
- Reflection (Class, Method) allows runtime introspection and invocation.

4) JVM concepts & memory (brief)

- **JVM memory regions**: Heap (objects), Stack (frames/local variables), Metaspace (class metadata).
- GC: multiple collectors exist (G1, ZGC, Shenandoah) — pick/configure based on latency vs throughput.

- Monitor allocations with tools: jvisualvm, jcmd, jmap, jstack.
- **Common performance tips**: prefer primitive arrays for hot data, minimize allocations in tight loops, use StringBuilder for repeated string concat in loops, avoid synchronized hotspots (use concurrent collections or StampedLock/ReadWriteLock).

5) Modules (Java 9+) — module declaration

```java
// module-info.java
module com.example.app {
    requires java.sql;
    exports com.example.app.api;
}
```

- Module system enforces encapsulation across modules and controls which packages are exported.

6) Example: Non-blocking HTTP client (Java 11) & CompletableFuture

```java
import java.net.http.*;
import java.net.URI;
import java.util.concurrent.*;

public class HttpAsync {
    public static void main(String[] args) throws Exception {
        HttpClient client = HttpClient.newHttpClient();
        HttpRequest req = HttpRequest.newBuilder()
            .uri(URI.create("https://httpbin.org/get"))
            .build();

        CompletableFuture<HttpResponse<String>> f = client.sendAsync(req, HttpResponse.BodyHandlers.ofString());
        f.thenAccept(resp -> System.out.println("Status: " + resp.statusCode()))
         .exceptionally(ex -> { ex.printStackTrace(); return null; });

        // block to keep the example alive
        f.join();
    }
}
```

- HttpClient supports async requests and integrates with CompletableFuture.

Design & style tips (practical)

- Favor **immutability** where possible (immutable objects are easier to reason about).
- Use **interfaces** for APIs, **final** for classes/variables when mutation is undesired.
- Use meaningful names and small methods (single responsibility).
- Handle exceptions properly: wrap checked exceptions into domain-specific ones when crossing layers; don't swallow exceptions.
- Use **logging** (SLF4J + a backend) instead of System.out.println in real apps.
- Write unit tests (JUnit 5), and consider property testing for tricky code paths.

Common pitfalls & gotchas

- Comparing Strings with == (use .equals()).
- NullPointerException — prefer Objects.requireNonNull() where appropriate, or Optional for return types where absence is expected.
- Mutable static state — avoid or guard with concurrency primitives.
- Autoboxing — watch for NullPointerException or performance overhead (prefer primitives in tight loops).
- Overusing reflection — useful but bypasses compile-time checks and is slower.

Quick reference cheatsheet

- Class/method: public class X { public static void main(String[] args) { ... } }
- Generics: List<String> Map<K,V>
- Lambda: (x) -> x + 1
- Stream terminal ops: collect, forEach, reduce
- Concurrency: ExecutorService, CompletableFuture, ConcurrentHashMap
- I/O: try (Resource r = ...) { } (try-with-resources)
- Exceptions: checked (must declare) vs unchecked (runtime)
- Annotations: @Override, @Deprecated, custom with @Retention