SYNERGETICS
GET IT RIGHT

Java 21 New Features

Makarand Bhoir

Microsoft Partner | Silver Learning Gold Cloud Platform

Microsoft Cloud Solution Provider

Educate Advise Implement Manage

Agenda

SYNERGETICS
GET IT RIGHT

- Records
- Sealed classes
- Virtual Thread
- String Templates
- Switch expressions and pattern matching
- Text blocks
- Sequence collection
- Scope Values
- Unnamed classes and Instance main method

Records

Records are a special type of class that help avoid boilerplate code. They are considered “data carriers”.

Records are immutable and are final by default.

You cannot extend your custom record because records already (implicitly) extend from the Record class. This is similar to enums (which implicitly extend from Enum).

records

Records can have both static fields and static methods.

Records can have instance methods.

Records cannot have instance fields. All the instance fields are listed as “components” in the record declaration.

Records can implement interfaces.

records

Records are specified using a record declaration where you specify the “components” of the record.

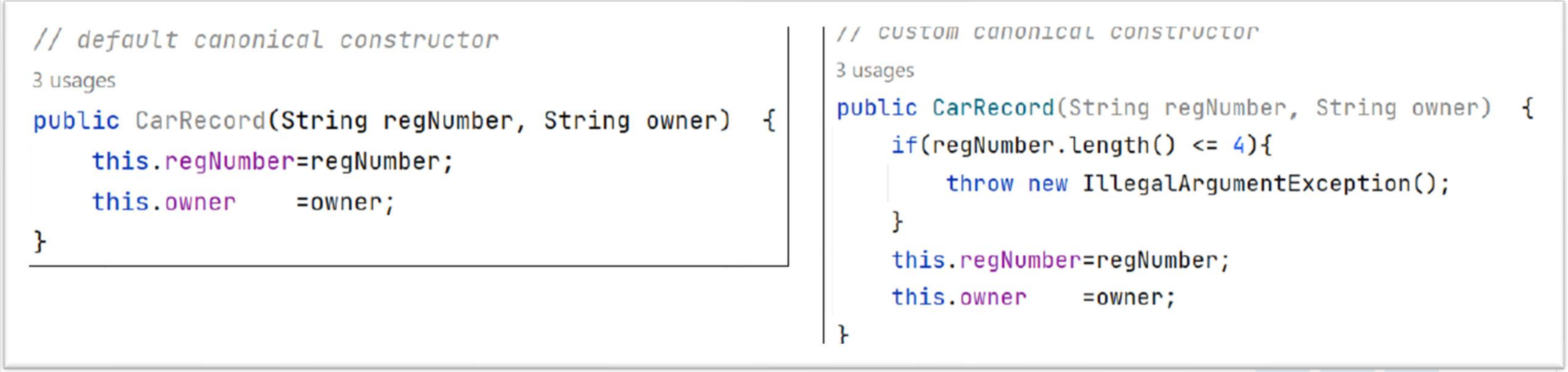
Implicitly generated are:

- canonical constructor
- toString () the string representation of all the record class's
- components , with their names
- equals() and hashCode() which specify that two record classes are equal if they are of the same type and contain equal component values
- public accessor methods with the same name as the components.

records

You can override all the default implementations. This includes the canonical constructor (perhaps for data validation).

Compact constructor is a concise variation of the canonical constructor and is specific to records.

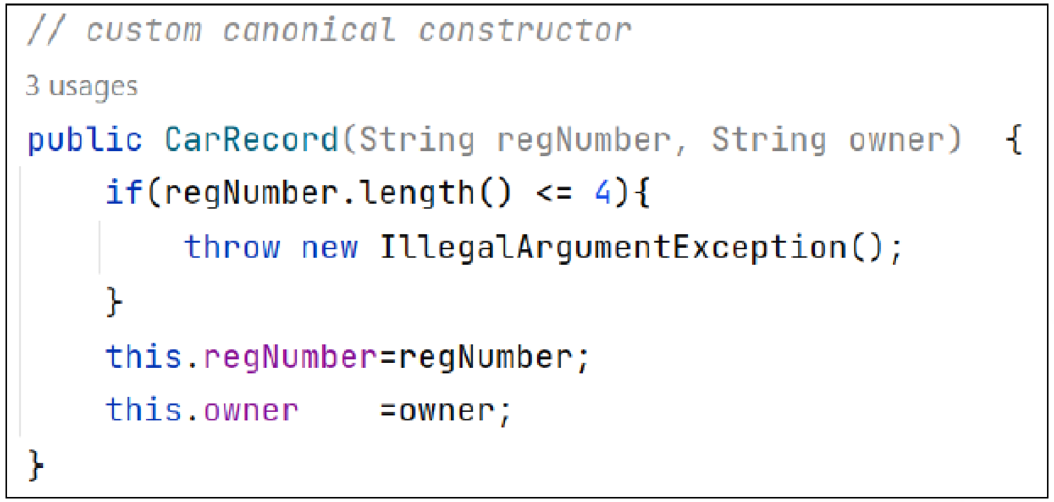


```
// default canonical constructor
3 usages
public CarRecord(String regNumber, String owner)  {
    this.regNumber=regNumber;
    this.owner    =owner;
}
```

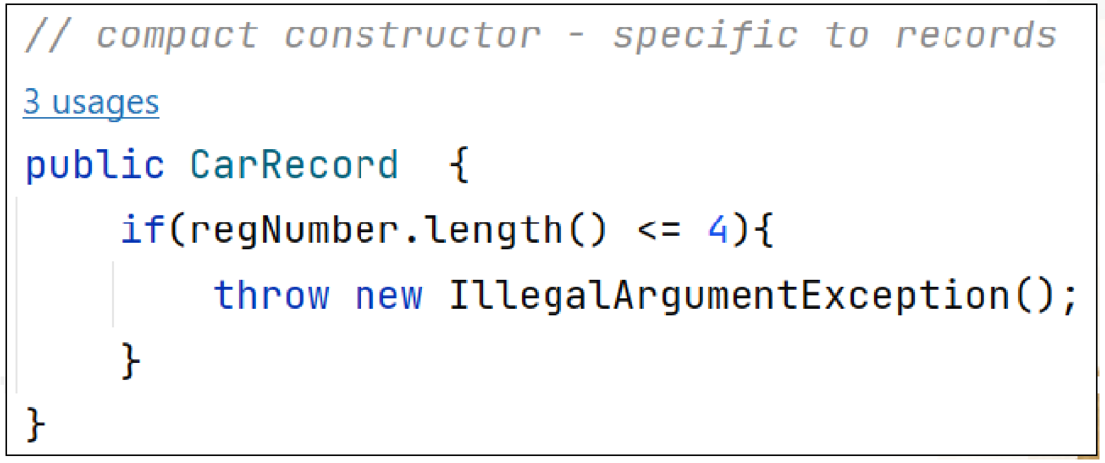
```
// custom canonical constructor
3 usages
public CarRecord(String regNumber, String owner)  {
    if(regNumber.length() <= 4){
        throw new IllegalArgumentException();
    }
    this.regNumber=regNumber;
    this.owner    =owner;
}
```

records

```
public record CarRecord(String regNumber, String owner)  {}
```



```
// custom canonical constructor
3 usages
public CarRecord(String regNumber, String owner)  {
    if(regNumber.length() <= 4){
        throw new IllegalArgumentException();
    }
    this.regNumber=regNumber;
    this.owner    =owner;
}
```



```
// compact constructor - specific to records
3 usages
public CarRecord  {
    if(regNumber.length() <= 4){
        throw new IllegalArgumentException();
    }
}
```

SYNERGETICS
GET IT RIGHT

Demo

Record

Educate | Advise | Implement | Manage

Sealed classes

Inheritance enables any class to inherit from any other class.

Making a class final prevents any class from inheriting from that class i.e., the final class cannot become a super type at all.

What if you wanted your class to be available for inheritance but only to certain classes?

Sealed classes

Sealed classes enable us to control the scope of inheritance by enabling us to specify a classes’ subtypes.

Also works with interfaces (we can define what classes implement the interface).

SYNERGETICS
GET IT RIGHT

Demo

Sealed

Educate Advise Implement Manage

Virtual Thread

SYNERGETICS
GET IT RIGHT

Virtual threads (JEP-425) are JVM-managed lightweight threads

It help in writing high-throughput concurrent applications

In contrast to platform threads, the virtual threads are not wrappers of OS threads

They are lightweight Java entities (with their own stack memory with a small footprint – only a few hundred bytes) that are cheap to create, block, and destroy

We can create many of them at the same time (millions) so they sustain a massive throughput.

Virtual Thread

SYNERGETICS
GET IT RIGHT

Virtual threads are stored in the JVM heap instead of the OS stack

The JVM schedules virtual threads to run on platform threads.

A platform thread executes only one virtual thread at a time, meaning there's a 1-to-1 mapping at any given moment.

However, virtual threads are not permanently tied to a platform thread.

When a virtual thread blocks (e.g., waiting for I/O), the JVM detaches it from the platform thread so that another virtual thread can use the platform thread.

Classic Platform Thread

Java has treated the platform threads as thin wrappers around operating system (OS) threads. Creating such platform threads has always been costly

Due to a large stack and other resources that are maintained by the operating system

The number of platform threads also has to be limited because these resource-hungry threads can affect the performance of the whole machine

This is mainly because platform threads are mapped 1:1 to OS threads

Scalability Issues with Platform Threads

Platform threads have always been easy to model, program and debug because they use the platform's unit of concurrency to represent the application's unit of concurrency. It is called thread-per-request pattern.

This pattern limits the throughput of the server because the number of concurrent requests (that server can handle) becomes directly proportional to the server's hardware performance.

Apart from the number of threads, latency is also a big concern

In today's world of microservices, a request is served by fetching/updating data on multiple systems and servers

While the application waits for the information from other servers, the current platform thread remains in an idle state

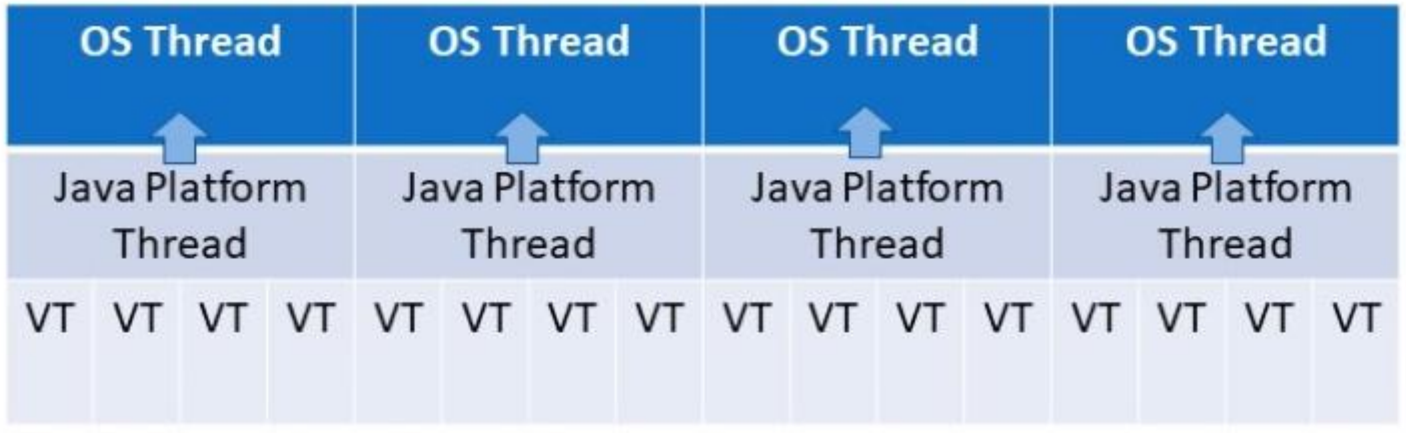
This is a waste of computing resources and a major hurdle in achieving a high throughput application

Virtual Threads Looks Promising

Similar to traditional threads, a virtual thread is also an instance of java.lang.Thread

Virtual thread runs its code on an underlying OS thread, but it does not block the OS thread for the code's entire lifetime

Keeping the OS threads free means that many virtual threads can run their Java code on the same OS thread, effectively sharing it.



* VT – Virtual Thread

Virtual Threads Looks Promising

Similar to traditional threads, the application's code runs in a virtual thread for the entire duration of a request (in thread-per-request style)

But the virtual thread consumes an OS thread only when it performs the calculations on the CPU.

They do not block the OS thread while they are waiting or sleeping.

Virtual threads help in achieving the same high scalability and throughput as the asynchronous APIs with the same hardware configuration, without adding the syntax complexity.

Virtual threads are best suited to executing code that spends most of its time blocked, waiting for data to arrive on a network socket or waiting for an element in queue for example.

Difference between Platform Threads and Virtual Threads

SYNERGETICS
GET IT RIGHT

Virtual threads are always daemon threads. The Thread.setDaemon(false) method cannot change a virtual thread to be a non-daemon thread.

Virtual threads always have the normal priority and the priority cannot be changed, even with setPriority(n) method. Calling this method on a virtual thread has no effect.

Virtual threads do not support the stop(), suspend(), or resume() methods. These methods throw an UnsupportedOperationException when invoked on a virtual thread.

SYNERGETICS
GET IT RIGHT

Demo

Virtual Thread

Educate Advise Implement Manage

Pattern matching

The switch statement has undergone several changes over the years.

switch expressions were introduced as a “preview feature” in Java 12 and became permanent in Java 14.

a “preview feature” is a feature that is designed and implemented but not yet permanent (and may never be); it allows a large developer audience to test out the feature before committing to it.

yield statement introduced in Java 13 to support switch expressions.

Pattern matching

This is a “preview feature” in Java 17.

Preview features are, by default disabled so you must explicitly enable them.

SYNERGETICS
GET IT RIGHT

Demo

Pattern matching

Educate Advise Implement Manage

Text Block

A text block is a String object and as a result, shares the same properties as String objects (immutable and interned).

you can call String methods on a text block.

A text block begins with three double quote characters followed by newline

- text blocks cannot be put on one line

Text Block Example

- TextBlockExample.java

Sequenced Collections

Sequenced Collections (JEP 431) establish new interfaces for collections with a defined encounter order.

Motivation:

- regarding defined encounter order, Java’s collections framework is neither straightforward or consistent.
- for example, List and Deque define an encounter order but their common supertype, Collection , does not
- Set does not define an encounter order, and neither does its subtype HastSet ; but other subtypes do, such as SortedSet (sorted) and LinkedHashSet (insertion order)

Sequenced Collections

Without interfaces to define them, operations related to encounter order are implemented inconsistently.

Even though many implementations support getting the first or last element, each collection defines its own way.

	First Element	Last Element
List	list.get(0)	list.get(list.size() -1)
Deque	deque.getFirst()	deque.getLast()
SortedSet	sortedSet.first()	sortedSet.last
LinkedHashSet	linkedHashSet.iterator().next()	// missing

Sequenced Collections

Iterating the elements in a collection from first to last is fine.

- for example: enhanced for loop, use an iterator or stream()

However, iterating in reverse order is a different matter.

- NavigableSet provides a descendingSet
- Deque provides descendingIterator
- List provides a ListIterator
- LinkedHashSet provides no support (one must copy its elements into another collection).

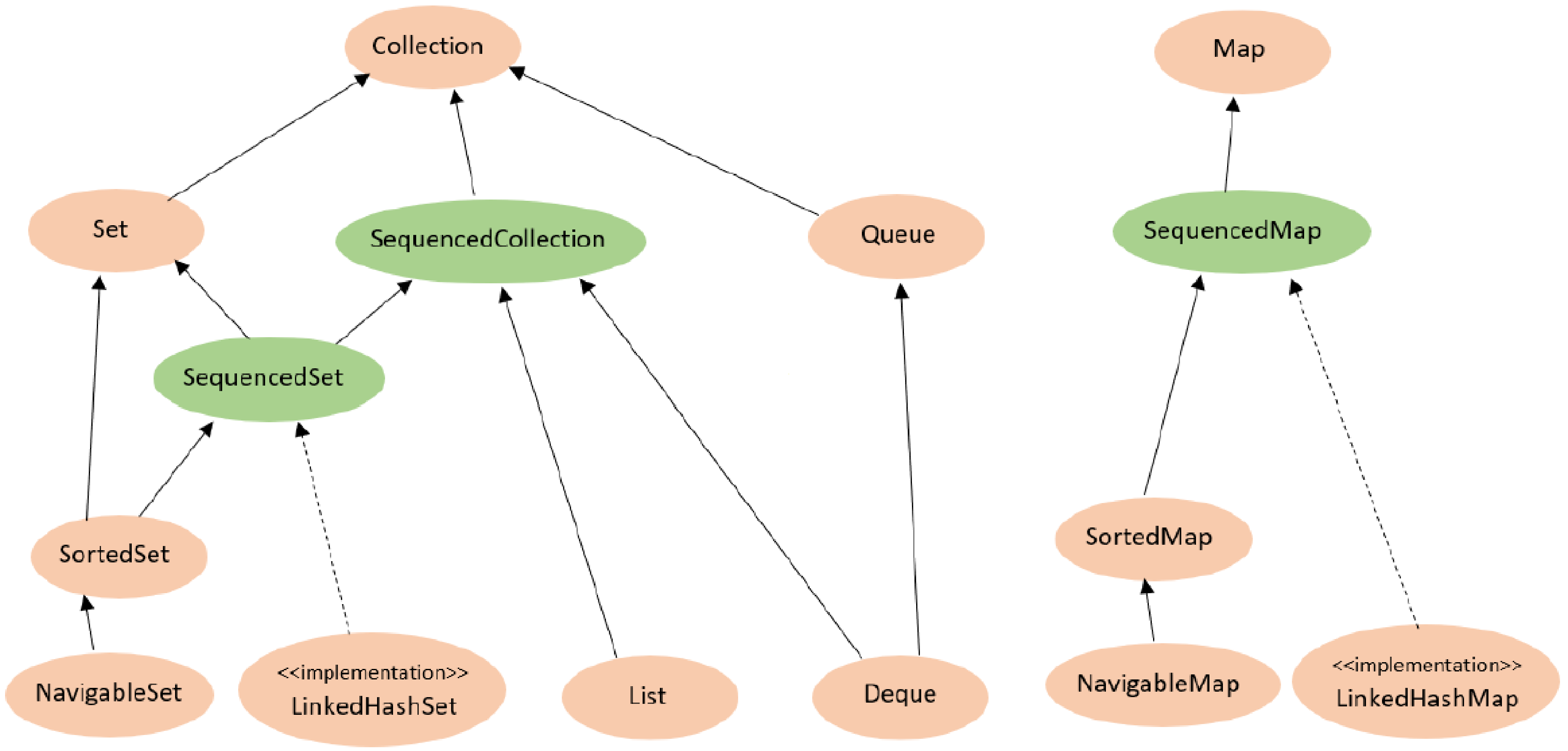
Sequenced Collections

In JEP 431, new interfaces are defined for sequenced collections, sequenced sets and sequenced maps.

APIs are defined for accessing first/last elements and also for processing elements in reverse order.

All of the methods in the interfaces have default implementations.

Sequenced Collections



Sequenced Collections API

```
interface SequencedCollection<E> extends Collection<E>{
     // new method
     SequencedCollection<E> reversed();
     // methods promoted from Deque
     void addFirst(E);
     void addLast(E);
     E getFirst();
     E getLast();
     E removeFirst();
     E removeLast();
}

interface SequencedSet<E> extends Set<E>, SequencedCollection <E>{
     // new method
     SequencedSet<E> reversed();
}
```

Sequenced Collections API

```
interface SequencedMap<K,V> extends Map<K,V>{
      // new methods
      SequencedMap<K,V> reversed();
      SequencedSet<K> sequencedKeySet();
      SequencedCollection<V> sequencedValues();
      SequencedSet<Entry<K,V>> sequencedEntrySet();
      V putFirst(K, V);
      V putLast(K, V);
      // methods promoted from NavigableMap
      Entry<K,V> firstEntry();
      Entry<K,V> lastEntry();
      Entry<K,V> pollFirstEntry();
      Entry<K,V> pollLastEntry();
}
```

SYNERGETICS
GET IT RIGHT

Demo

Sequenced Collections API

Educate Advise Implement Manage

SYNERGETICS
GET IT RIGHT

Unnamed classes and Instance main method

Java supports both “programming in the small” (variables, methods, control flow etc..) and “programming in the large” (classes, interfaces, packages, modules

The goal is to focus on the “programming in the small” by reducing ceremony/scaffolding for those learning the language.

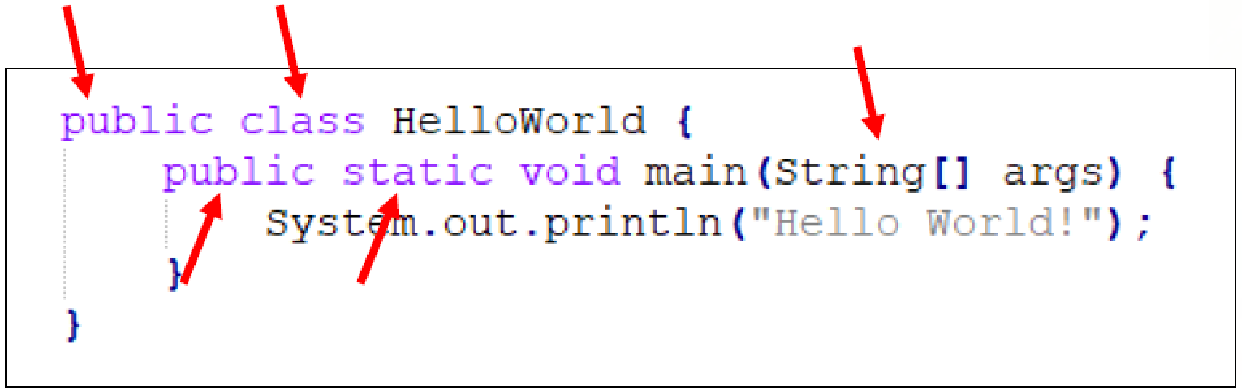
Constructs such as classes, access modifiers such as public and keywords such as static relate to “programming in the large” and should only be encountered when required.

Educate Advise Implement Manage

Unnamed classes and Instance main method

In effect, make Java easier to learn. To this end, JEP 445 enables learners to write their first programs without needing to understand language features designed for large programs.

Basic programs in a concise manner.

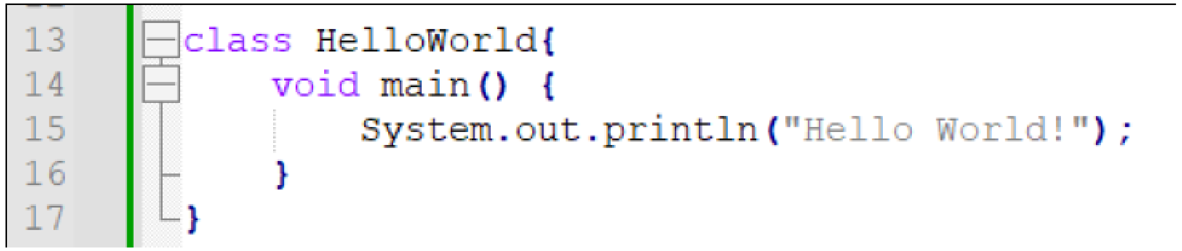


```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Instance main method

No need for static, public or a String [] parameter

If you have both the traditional public static void main(String[] args and the instance main(), the traditional version takes precedence.



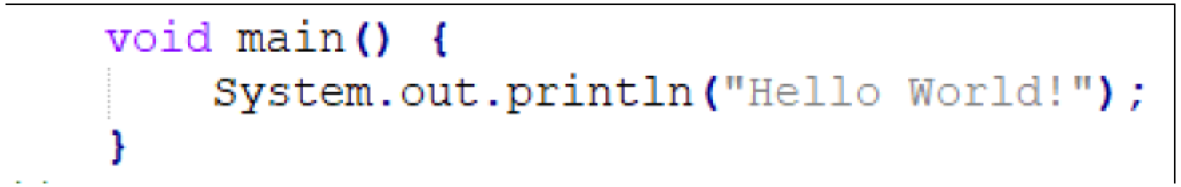
```
class HelloWorld{
    void main() {
        System.out.println("Hello World!");
    }
}
```

Unnamed Classes

extend from Object and cannot implement an interface

are final and reside in the unnamed package

The .class name on the hard disk depends on the filename for example, if the above code is in HelloWorld.java , HelloWorld.class is created on the hard disk



```
void main() {
    System.out.println("Hello World!");
}
```

Unnamed Classes

are exactly like normal classes except that an unnamed class has only one constructor the default no args constructor provided by the compiler.

it is an error to explicitly code a constructor, even a no args constructor.

the this keyword is still valid.

Unnamed Classes

as code cannot refer to an unnamed class by name, instances of an unnamed class cannot be constructed directly.

therefore, such classes are useful for standalone programs or as an entry point to a program.

as a result, unnamed classes must have a main() method.

SYNERGETICS
GET IT RIGHT

Demo

Unnamed classes and Instance main method

Educate Advise Implement Manage

SYNERGETICS
GET IT RIGHT

Thank You

Educate Advise Implement Manage