UNIVERSITÀ DI PARMA
il mondo che ti aspetta
DIPARTIMENTO DI INGEGNERIA E ARCHITETTURA

Distributed Systems

Java Evolution: from Java 1.8 onwards

Prof. Agostino Poggi

Java SE Releases

From Java SE 9 Oracle JDK and OpenJDK implementations

JDK 1.0 - January 1996	Java SE 9 - September 2017
JDK 1.1 - February 1997	Java SE 10 - March 2018
J2SE 1.2 - December 1998	Java SE 11 - September 2018 LTS
J2SE 1.3 - May 2000	Java SE 12 - March 2019
J2SE 1.4 - February 2002	Java SE 13 - September 2019
J2SE 5.0 - September 2004	Java SE 14 - March 2020
Java SE 6 - December 2006	Java SE 15 - September 2020
Java SE 7 - July 2011	Java SE 16 - March 16th, 2021
Java SE 8 - March 2014 LTS	Java SE 17 - September 2021 LTS

Long-Term Support

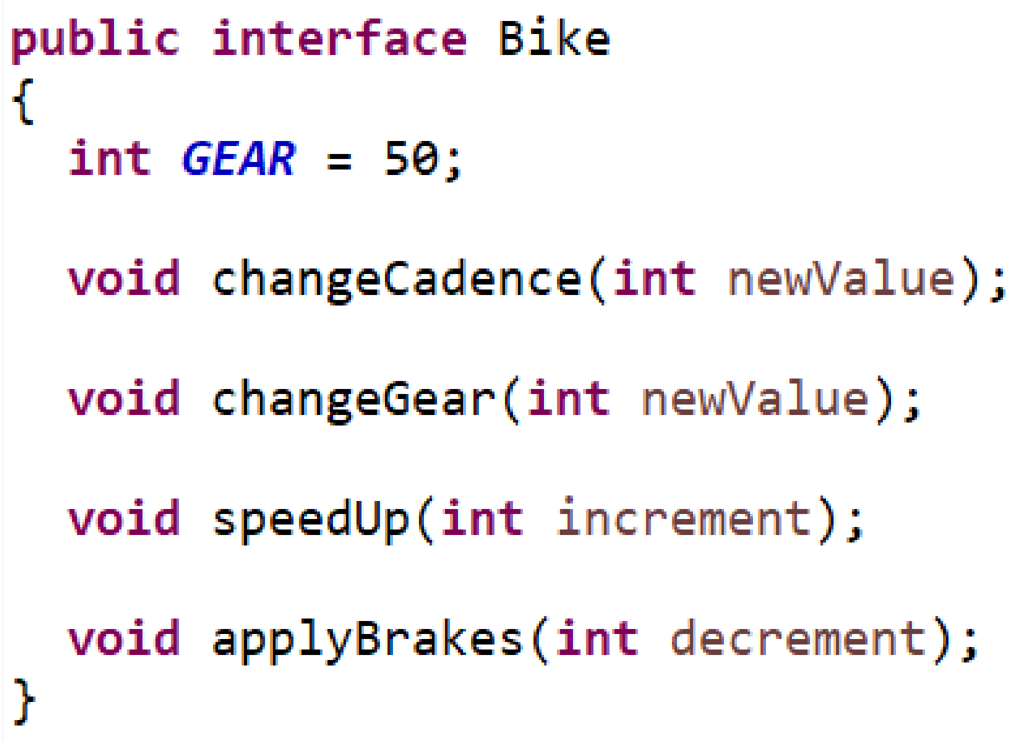
LTS releases are every three years

Java Interfaces

In the initial implementation of Java, an interface groups a set of **abstract methods**, i.e., methods with empty bodies

Moreover, such kind of interface can contain some **static constants**

Both methods and static constants must be **public**



```
public interface Bike
{
  int GEAR = 50;

  void changeCadence(int newValue);

  void changeGear(int newValue);

  void speedUp(int increment);

  void applyBrakes(int decrement);
}
```

Interface Use Problems

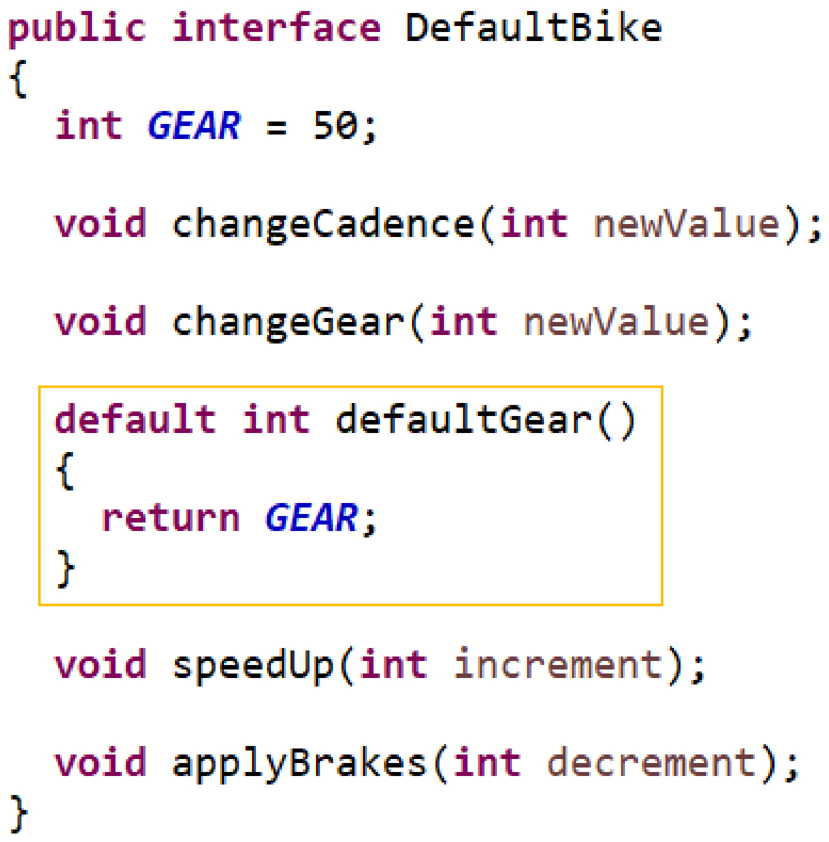
- If **different classes** must implements the **same interface** and the interface offers **only abstract** methods, then **each classes** must **implements** the **methods** of the interface even if all the classes use the **same implementation** for some of them

- **Default** and **static methods** cope with this problem

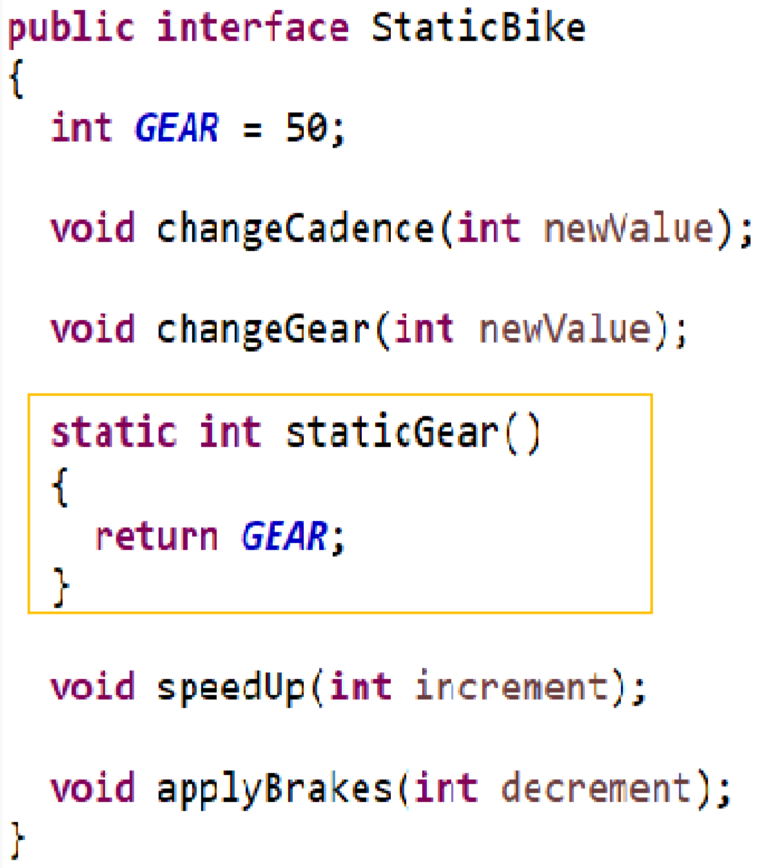
- Moreover, **classes** can **override** the **default methods**, but they **cannot override** the **static methods**

- The **multiple inheritance** problem can occur, when a class implements **two interfaces** with a **default methods** of **same signature**. In this case the class must **override** both two **default methods** proving a **new method** with the **same** signature

Java Interfaces – Default & Static Method

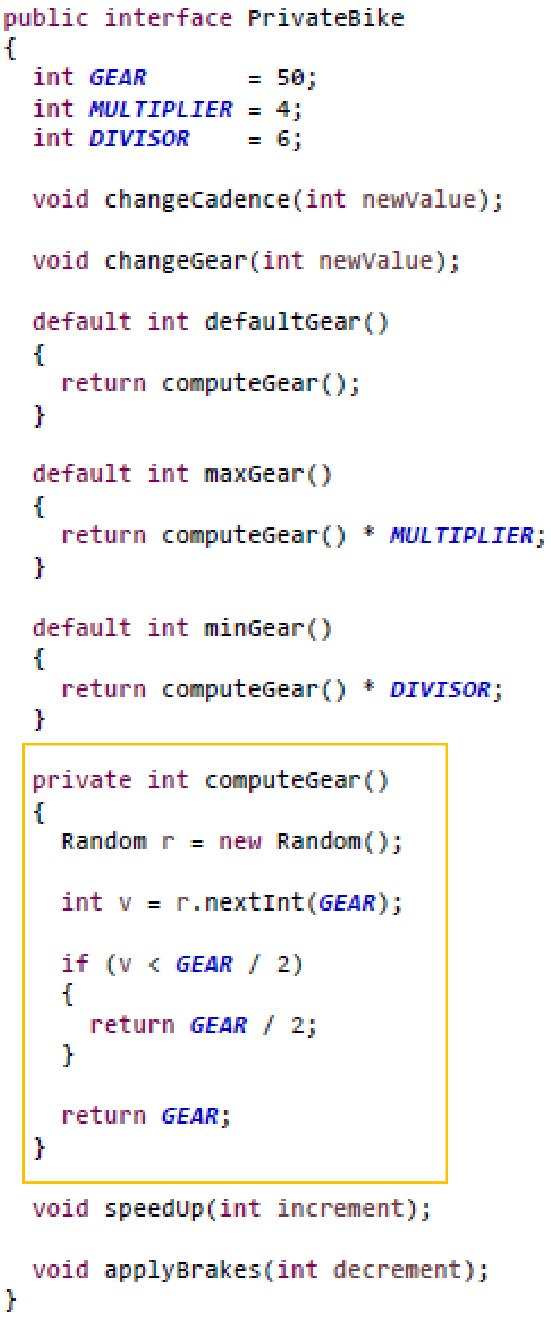


```
public interface DefaultBike
{
  int GEAR = 50;

  void changeCadence(int newValue);

  void changeGear(int newValue);

  default int defaultGear()
  {
    return GEAR;
  }

  void speedUp(int increment);

  void applyBrakes(int decrement);
}
```



```
public interface StaticBike
{
  int GEAR = 50;

  void changeCadence(int newValue);

  void changeGear(int newValue);

  static int staticGear()
  {
    return GEAR;
  }

  void speedUp(int increment);

  void applyBrakes(int decrement);
}
```

Java SE 8



```
public interface PrivateBike
{
  int GEAR       = 50;
  int MULTIPLIER = 4;
  int DIVISOR    = 6;

  void changeCadence(int newValue);

  void changeGear(int newValue);

  default int defaultGear()
  {
    return computeGear();
  }

  default int maxGear()
  {
    return computeGear() * MULTIPLIER;
  }

  default int minGear()
  {
    return computeGear() * DIVISOR;
  }

  private int computeGear()
  {
    Random r = new Random();

    int v = r.nextInt(GEAR);

    if (v < GEAR / 2)
    {
      return GEAR / 2;
    }

    return GEAR;
  }

  void speedUp(int increment);

  void applyBrakes(int decrement);
}
```

Private methods improve code re-usability inside interfaces

Moreover, they expose to the user only the intended methods implementation

Java SE 9

Modular Programming and Java

- Usually, **complex systems** are **built** by **decomposing** such systems in **modules**
- A good decomposition in modules a good definition of the system modules should allow that
  - Each **module** can be **written** with **little knowledge** of code of the **other modules**
  - Each **module** can be **replaced without reassembly** of the **whole system**
- A **package** provide a **logical namespace** for a **group** of **related classes**
- Java supports **modular decomposition** through **packages** and **modules**

Platform Module System

- Introduced by Java 9 in 2017 to add a **higher level** of **aggregation** above **packages** though the use of **modules**

- A **module** defines a **uniquely named**, reusable group of related **packages**, as well as **resources** (such as images and XML files)

- A **module** needs a **Java module descriptor** named **module-info.java** which has to be located in the corresponding **module root directory** (package)

- A **module** can **depend** by **other modules**, but its **dependency graph** must be an **acyclic graph**

- A **package** must **be** in a **single module**

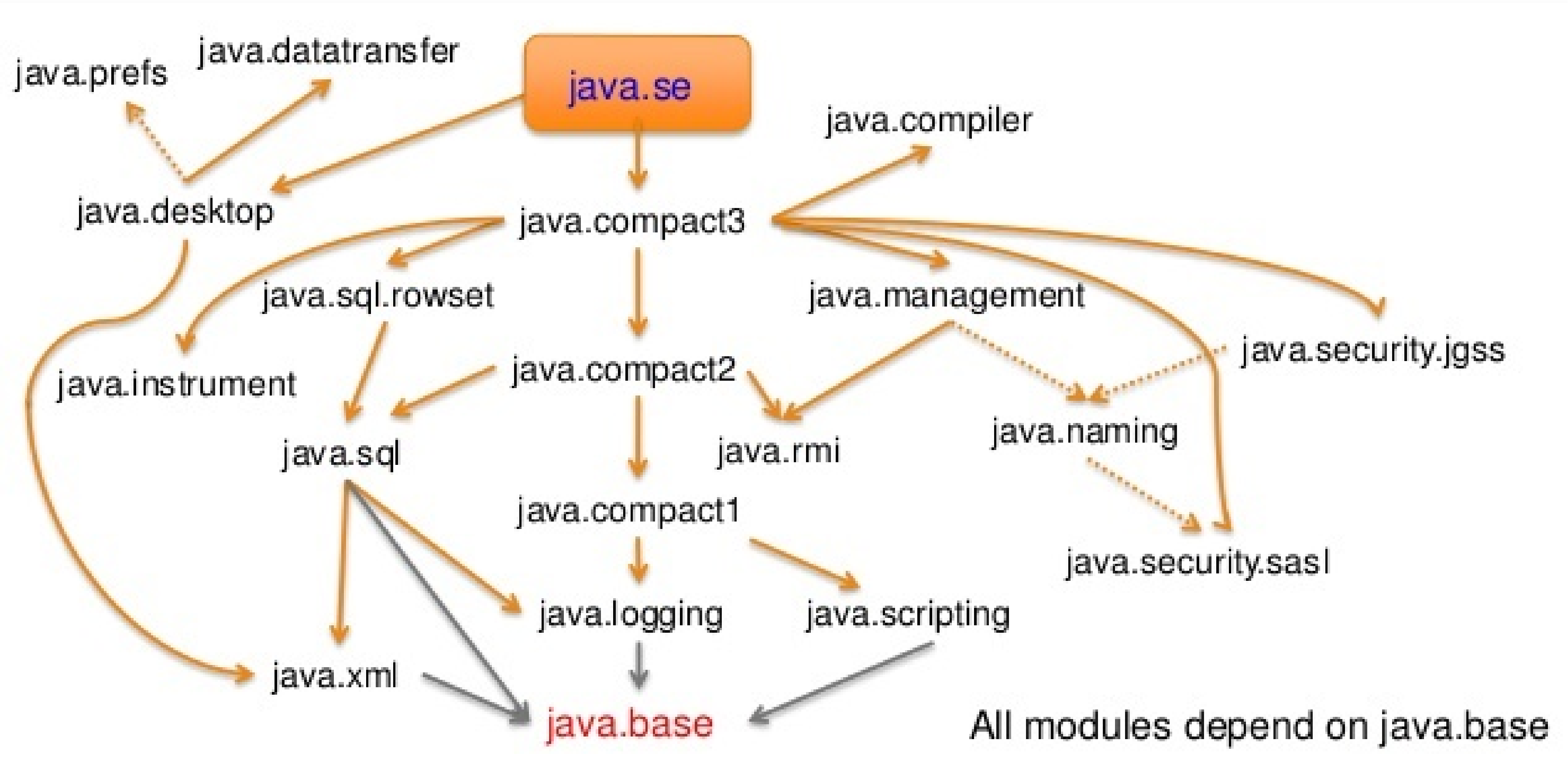
Main Features

- Reliable configuration
  - Modularity provides mechanisms for **explicitly declaring dependencies** between **modules**
- Strong encapsulation
  - **Packages** in a module **are accessible** to **other modules** only if the **module explicitly exports**
- Scalable Java platform
  - Java platform involves a **lot of modules**, but **custom runtimes** can be built using only the **modules needed** for the different **applications**

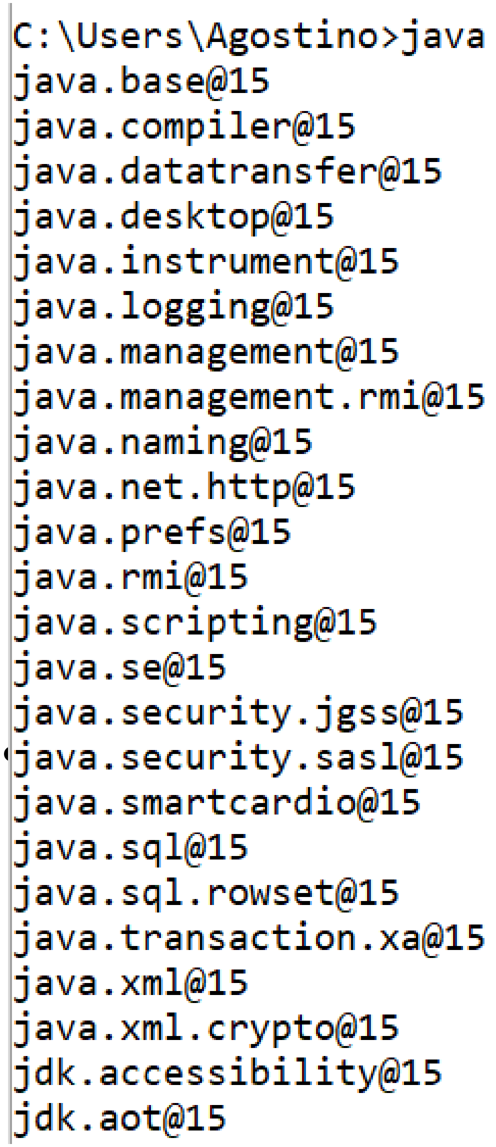
Module Descriptor

- Module **name**
- Module **dependencies**
  - I.e., modules from which it depends on
- **Exported** packages
  - I.e., packages in that are available to the other modules
- **Provided** services
  - I.e., services offered to other modules
- **Used** services
  - I.e., services provided by other modules
- **Reflection enabled** modules
  - I.e., modules that can use reflection to access it

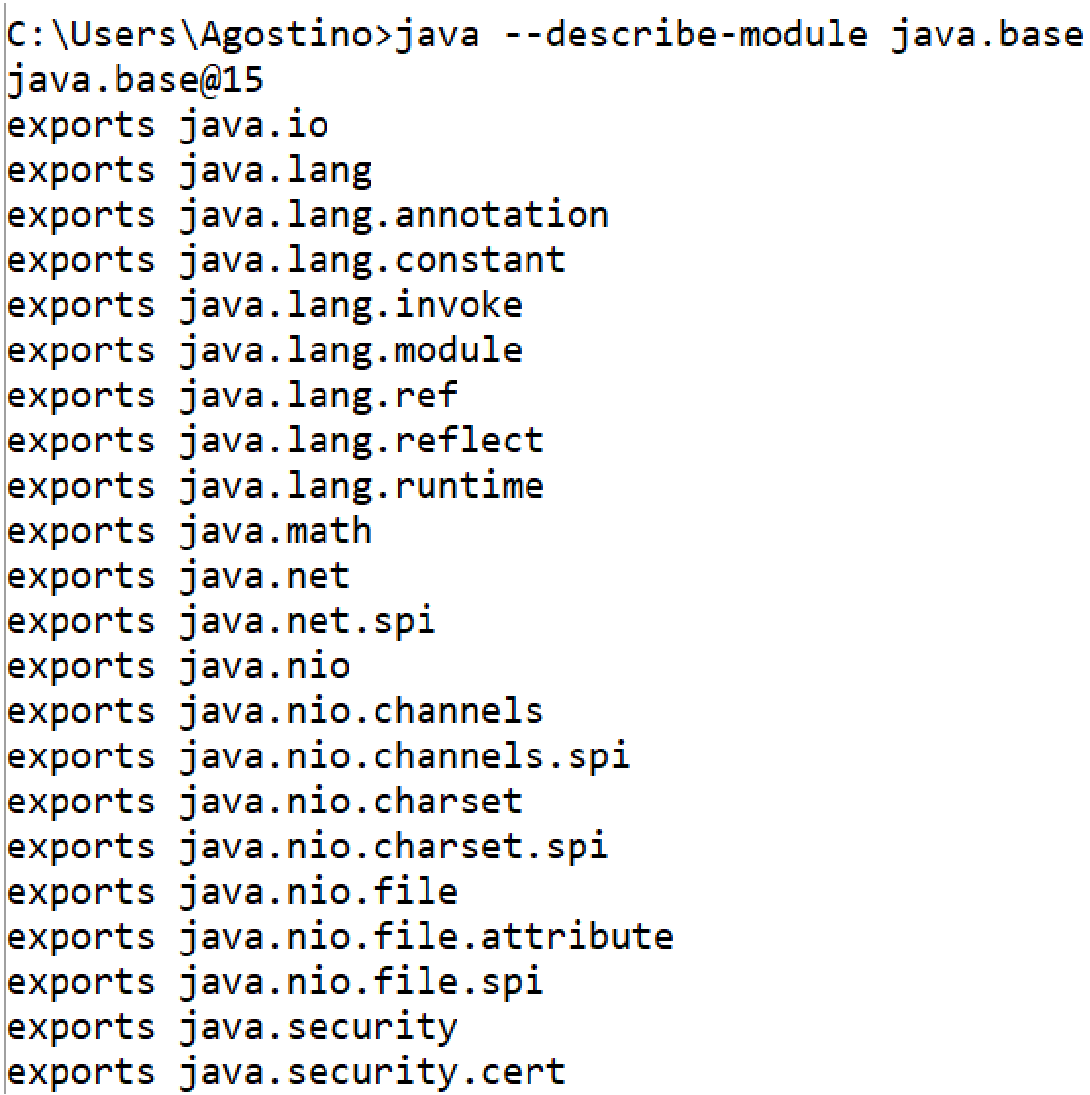
Java 9 Main Platform Modules



Explore Java Modules

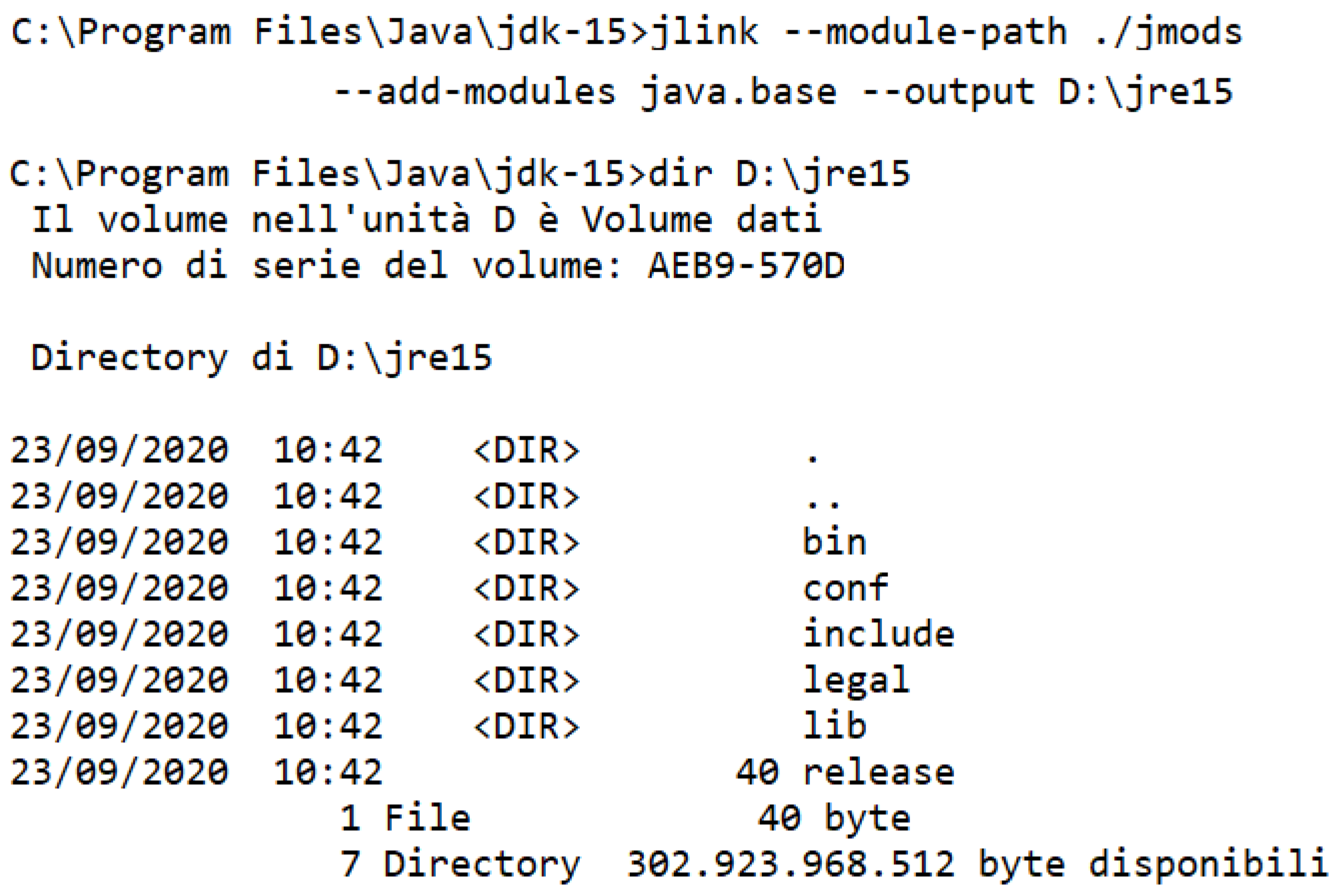


```
C:\Users\Agostino>java --list-modules
java.base@15
java.compiler@15
java.datatransfer@15
java.desktop@15
java.instrument@15
java.logging@15
java.management@15
java.management.rmi@15
java.naming@15
java.net.http@15
java.prefs@15
java.rmi@15
java.scripting@15
java.se@15
java.security.jgss@15
java.security.sasl@15
java.smartcardio@15
java.sql@15
java.sql.rowset@15
java.transaction.xa@15
java.xml@15
java.xml.crypto@15
jdk.accessibility@15
jdk.aot@15
```



```
C:\Users\Agostino>java --describe-module java.base
java.base@15
exports java.io
exports java.lang
exports java.lang.annotation
exports java.lang.constant
exports java.lang.invoke
exports java.lang.module
exports java.lang.ref
exports java.lang.reflect
exports java.lang.runtime
exports java.math
exports java.net
exports java.net.spi
exports java.nio
exports java.nio.channels
exports java.nio.channels.spi
exports java.nio.charset
exports java.nio.charset.spi
exports java.nio.file
exports java.nio.file.attribute
exports java.nio.file.spi
exports java.security
exports java.security.cert
```

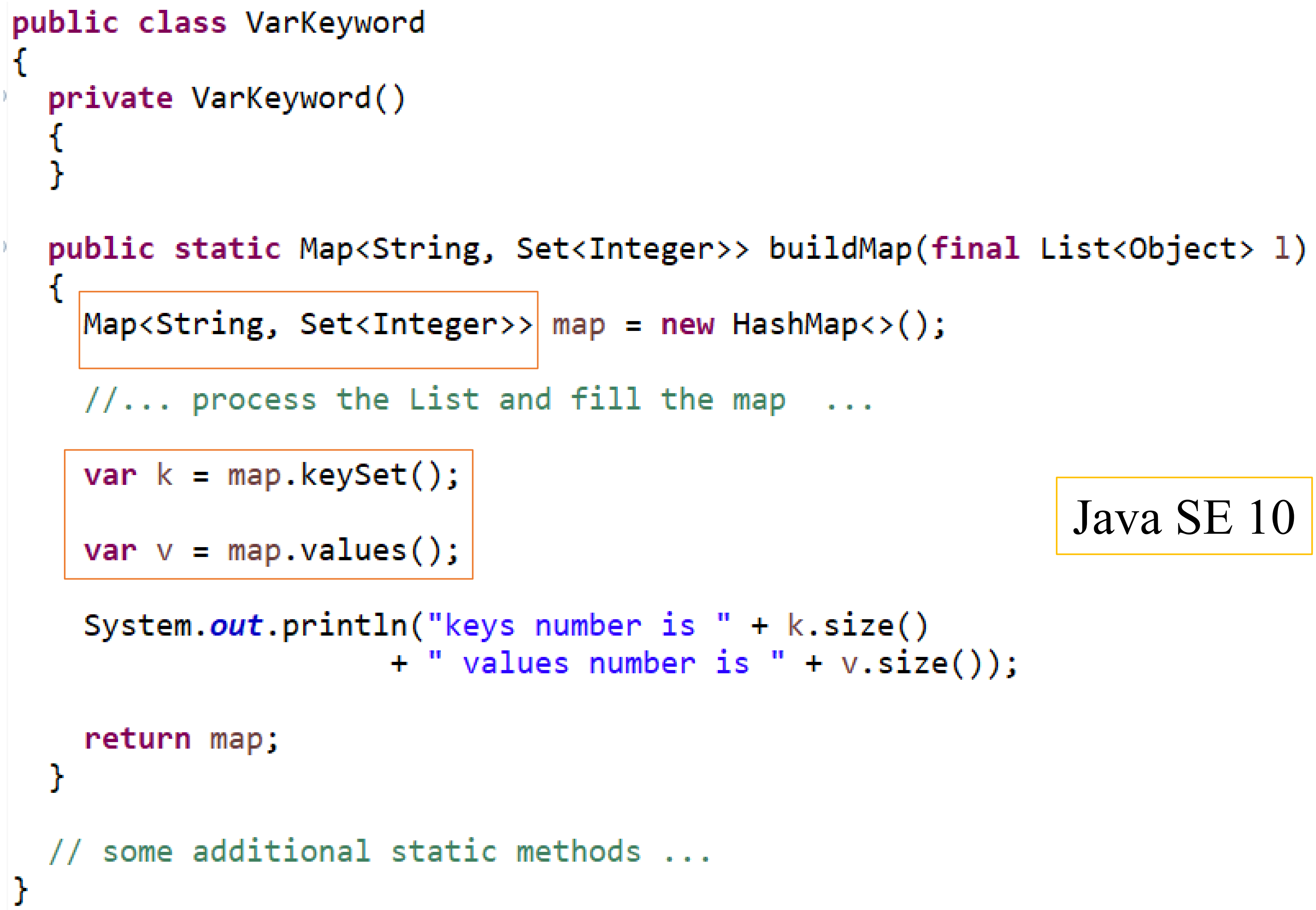
Build a Java JRE



```
C:\Program Files\Java\jdk-15>jlink --module-path ./jmods
               --add-modules java.base --output D:\jre15

C:\Program Files\Java\jdk-15>dir D:\jre15
 Il volume nell'unità D è Volume dati
 Numero di serie del volume: AEB9-570D

 Directory di D:\jre15

23/09/2020  10:42    <DIR>          .
23/09/2020  10:42    <DIR>          ..
23/09/2020  10:42    <DIR>          bin
23/09/2020  10:42    <DIR>          conf
23/09/2020  10:42    <DIR>          include
23/09/2020  10:42    <DIR>          legal
23/09/2020  10:42    <DIR>          lib
23/09/2020  10:42                40 release
               1 File             40 byte
               7 Directory  302.923.968.512 byte disponibili
```

Var Keyword – Type Inference

The **var** keyword asks the compiler to identify the type of the variables by using the surround context



```
public class VarKeyword
{
  private VarKeyword()
  {
  }

  public static Map<String, Set<Integer>> buildMap(final List<Object> l)
  {
    Map<String, Set<Integer>> map = new HashMap<>();

    //... process the List and fill the map  ...

    var k = map.keySet();

    var v = map.values();

    System.out.println("keys number is " + k.size()
                     + " values number is " + v.size());

    return map;
  }

  // some additional static methods ...
}
```

Java SE 10

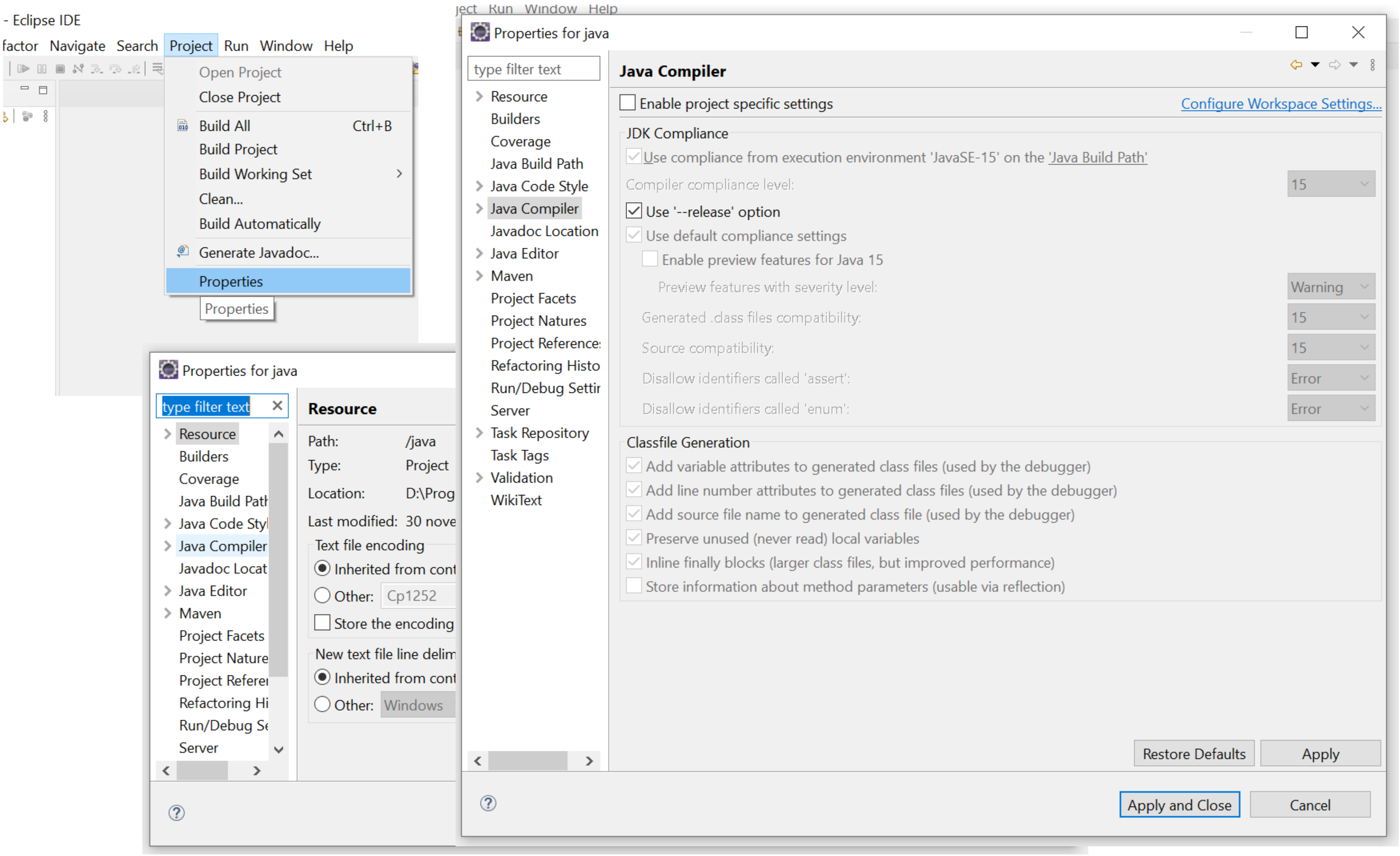
Java Releases Frequency Problem

- Java releases are delivered every **six months** and usually add new features
- It allows **less** waiting **time** for **new Java features**
- It allows **less time** to react to **feedback** about new features
- Therefore, a small **malfunction** in one implementation or a **poor feature** design could turn out to be **very costly**
- New Java features would benefit from a more **long period** of broad exposure and **evaluation**

Java Preview Features

- Java preview features are **completely specified** and **developed features** that are going through evaluation until they reach the final state
- Preview features are essentially just a way to **encourage** the community to **review** and provide **feedback**
  - Preview stage involves **more** than one **Java release**
  - However, **not** every Java **feature** must go through a **preview stage** in order to become final
- Preview features are **disabled** by **default**
  - To enable them, we must use the **enable-preview** argument, which enables all preview features at once
  - The **Java compiler**, as well as the **JVM,** must be of the **same Java version** that includes the **preview feature** we want to use

Enabling preview Features with Eclipse (1/2)



Resource

Path: /java
Type: Project
Location: D:\Prog
Last modified: 30 nove

Text file encoding
● Inherited from cont
○ Other: Cp1252
□ Store the encoding

New text file line delim
● Inherited from cont
○ Other: Windows

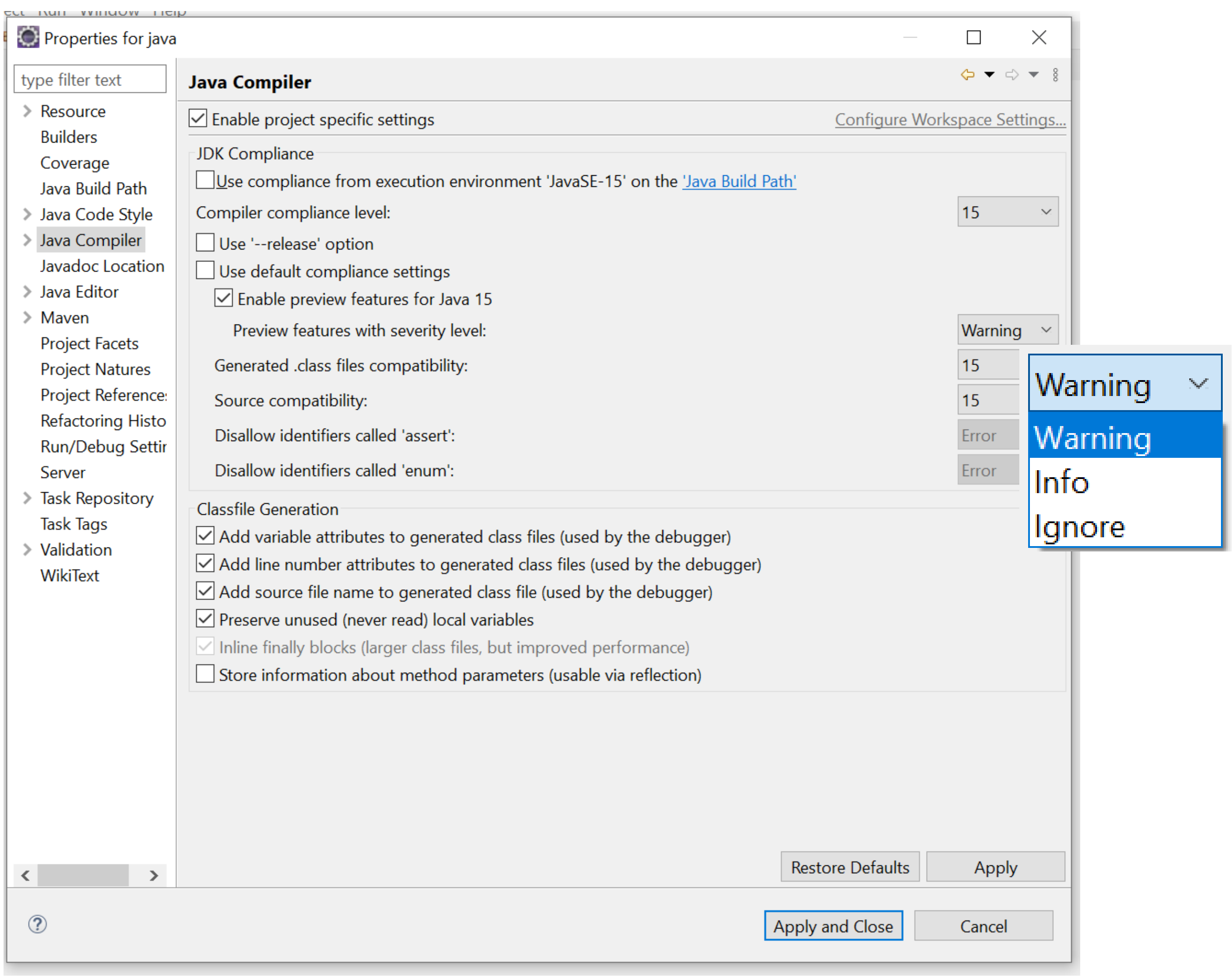
Java Compiler

□ Enable project specific settings   Configure Workspace Settings...

JDK Compliance
☑ Use compliance from execution environment 'JavaSE-15' on the 'Java Build Path'
Compiler compliance level: 15
☑ Use '--release' option
☑ Use default compliance settings
□ Enable preview features for Java 15
Preview features with severity level: Warning
Generated .class files compatibility: 15
Source compatibility: 15
Disallow identifiers called 'assert': Error
Disallow identifiers called 'enum': Error

Classfile Generation
☑ Add variable attributes to generated class files (used by the debugger)
☑ Add line number attributes to generated class files (used by the debugger)
☑ Add source file name to generated class file (used by the debugger)
☑ Preserve unused (never read) local variables
☑ Inline finally blocks (larger class files, but improved performance)
□ Store information about method parameters (usable via reflection)

Enabling preview Features with Eclipse (2/2)



Java Compiler

☑ Enable project specific settings — Configure Workspace Settings...

JDK Compliance

☐ Use compliance from execution environment 'JavaSE-15' on the 'Java Build Path'

Compiler compliance level: 15

☐ Use '--release' option

☐ Use default compliance settings

☑ Enable preview features for Java 15

Preview features with severity level: Warning

Generated .class files compatibility: 15

Source compatibility: 15

Disallow identifiers called 'assert': Error

Disallow identifiers called 'enum': Error

Classfile Generation

☑ Add variable attributes to generated class files (used by the debugger)

☑ Add line number attributes to generated class files (used by the debugger)

☑ Add source file name to generated class file (used by the debugger)

☑ Preserve unused (never read) local variables

☑ Inline finally blocks (larger class files, but improved performance)

☐ Store information about method parameters (usable via reflection)

Switch Expressions (1/3)

```java
final int size = 7;

Random random = new Random();

Day day = Day.values()[random.nextInt(size)];
```

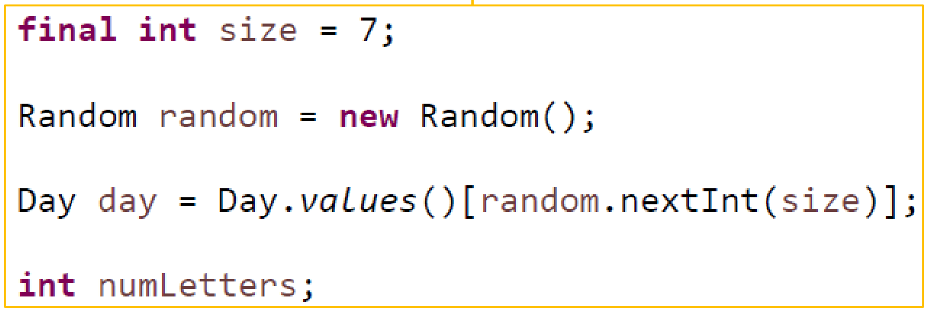
```java
switch (day)
{
  case MONDAY:
  case FRIDAY:
  case SUNDAY:
    System.out.println(day + " word letters are " + 6);
    break;
  case TUESDAY:
    System.out.println(day + " word letters are " + 7);
    break;
  case THURSDAY:
  case SATURDAY:
    System.out.println(day + " word letters are " + 8);
    break;
  case WEDNESDAY:
    System.out.println(day + " word letters are " + 9);
    break;
}

day = Day.values()[random.nextInt(size)];

switch (day)
{
  case MONDAY, FRIDAY, SUNDAY -> System.out.println(
      day + " word letters are " + 6);
  case TUESDAY -> System.out.println(day + " word letters are " + 7);
  case THURSDAY, SATURDAY -> System.out.println(
      day + " word letters are " + 8);
  case WEDNESDAY -> System.out.println(day + " word letters are " + 9);
}
```

day value can be: char, int, byte, short, their object wrappers, String and enum

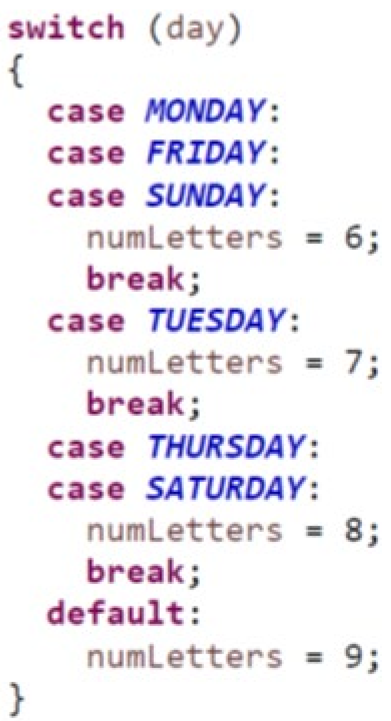
Java SE 12

1st Preview

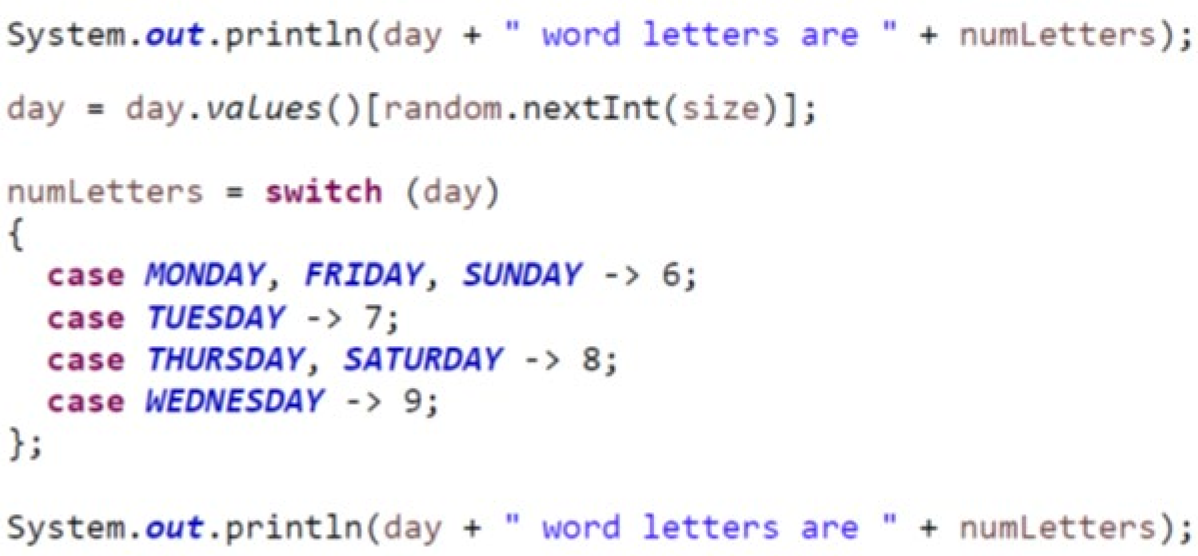
Switch Expressions (2/3)



```
final int size = 7;

Random random = new Random();

Day day = Day.values()[random.nextInt(size)];

int numLetters;
```



```
switch (day)
{
  case MONDAY:
  case FRIDAY:
  case SUNDAY:
    numLetters = 6;
    break;
  case TUESDAY:
    numLetters = 7;
    break;
  case THURSDAY:
  case SATURDAY:
    numLetters = 8;
    break;
  default:
    numLetters = 9;
}
```

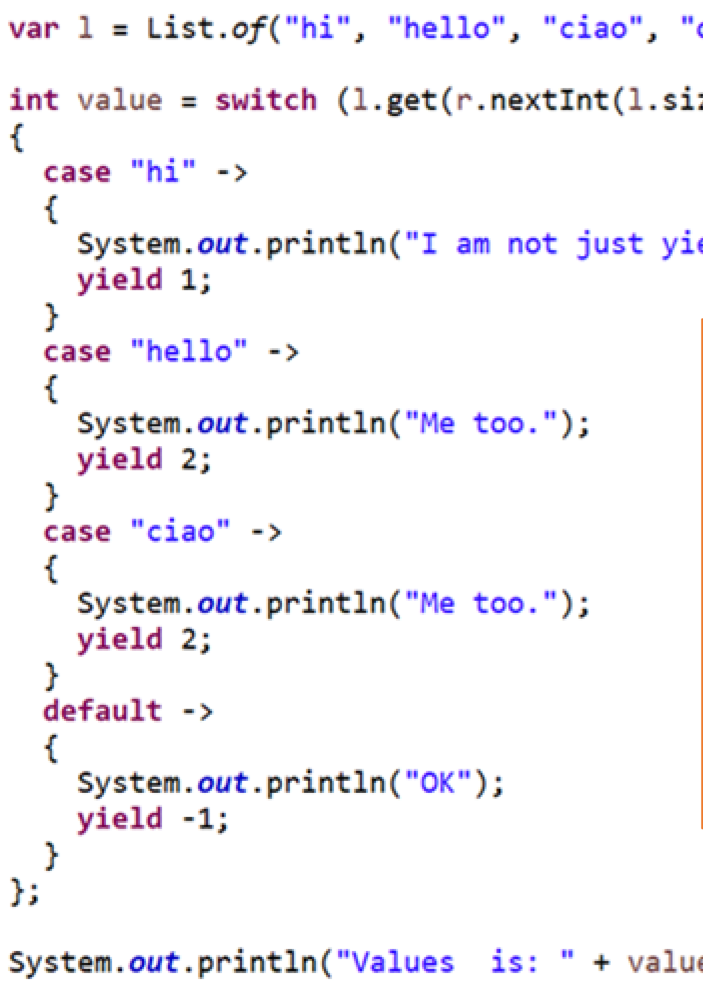


```
System.out.println(day + " word letters are " + numLetters);

day = day.values()[random.nextInt(size)];

numLetters = switch (day)
{
  case MONDAY, FRIDAY, SUNDAY -> 6;
  case TUESDAY -> 7;
  case THURSDAY, SATURDAY -> 8;
  case WEDNESDAY -> 9;
};

System.out.println(day + " word letters are " + numLetters);
```

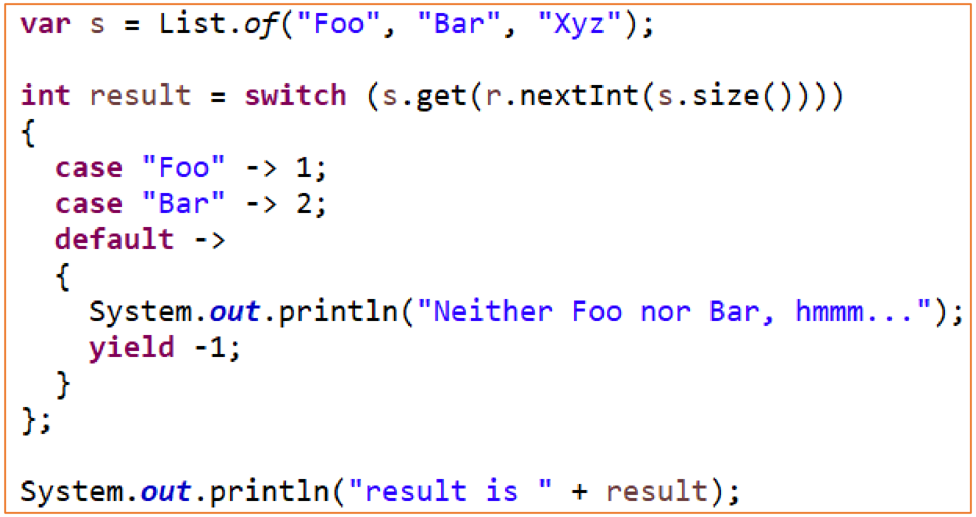
Java SE 12

1st Preview

Switch Expressions (3/3)



```
var l = List.of("hi", "hello", "ciao", "ok");

int value = switch (l.get(r.nextInt(l.size())))
{
  case "hi" ->
  {
    System.out.println("I am not just yielding!");
    yield 1;
  }
  case "hello" ->
  {
    System.out.println("Me too.");
    yield 2;
  }
  case "ciao" ->
  {
    System.out.println("Me too.");
    yield 2;
  }
  default ->
  {
    System.out.println("OK");
    yield -1;
  }
};

System.out.println("Values  is: " + value);
```



```
var s = List.of("Foo", "Bar", "Xyz");

int result = switch (s.get(r.nextInt(s.size())))
{
  case "Foo" -> 1;
  case "Bar" -> 2;
  default ->
  {
    System.out.println("Neither Foo nor Bar, hmmm...");
    yield -1;
  }
};

System.out.println("result is " + result);
```

Java SE 13

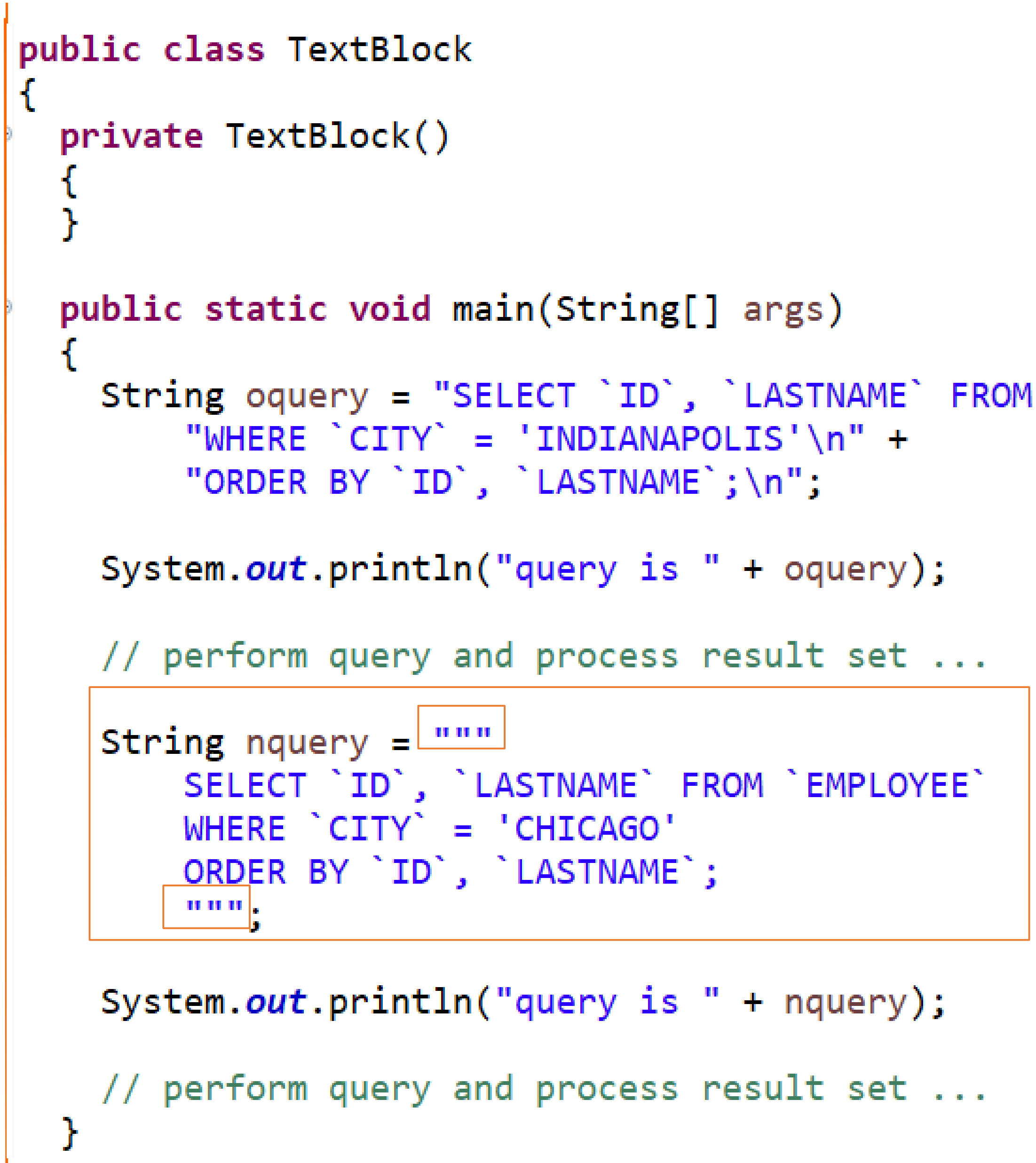
2nd Preview

Java SE 14

Permanent

The keyword **yield** is ever used to specify a **return value** from inside a **switch statement**. It is **different** to a **return** as it **yields from** a **statement** as opposed to **returns from** a **method**

Text Block



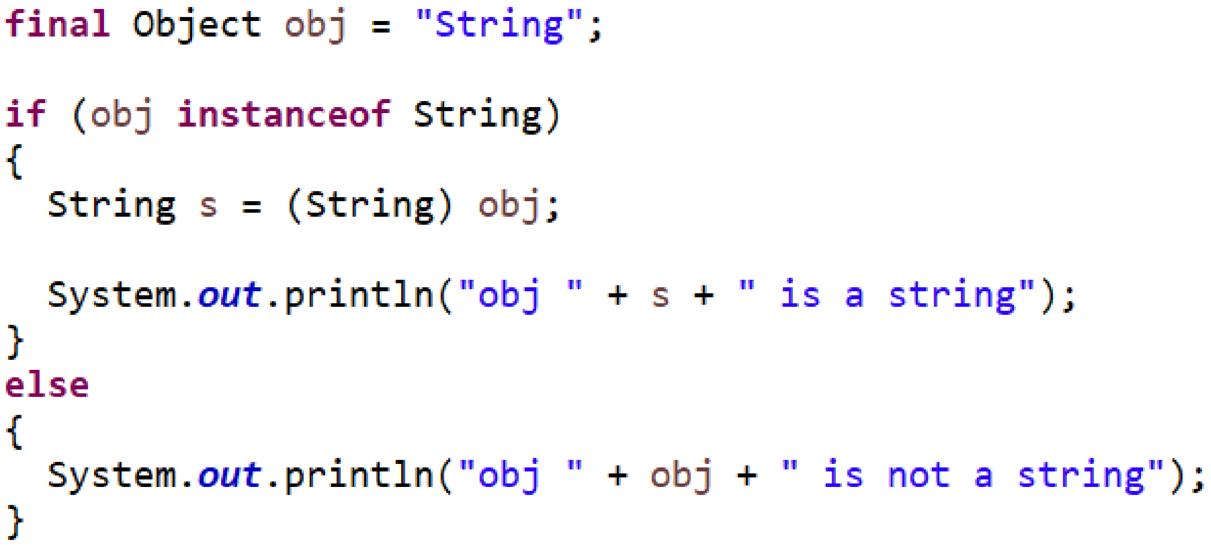
```
public class TextBlock
{
  private TextBlock()
  {
  }

  public static void main(String[] args)
  {
    String oquery = "SELECT `ID`, `LASTNAME` FROM `EMPLOYEE`\n" +
        "WHERE `CITY` = 'INDIANAPOLIS'\n" +
        "ORDER BY `ID`, `LASTNAME`;\n";

    System.out.println("query is " + oquery);

    // perform query and process result set ...

    String nquery = """
        SELECT `ID`, `LASTNAME` FROM `EMPLOYEE`
        WHERE `CITY` = 'CHICAGO'
        ORDER BY `ID`, `LASTNAME`;
        """;

    System.out.println("query is " + nquery);

    // perform query and process result set ...
  }
```

Java SE 13
1st Preview

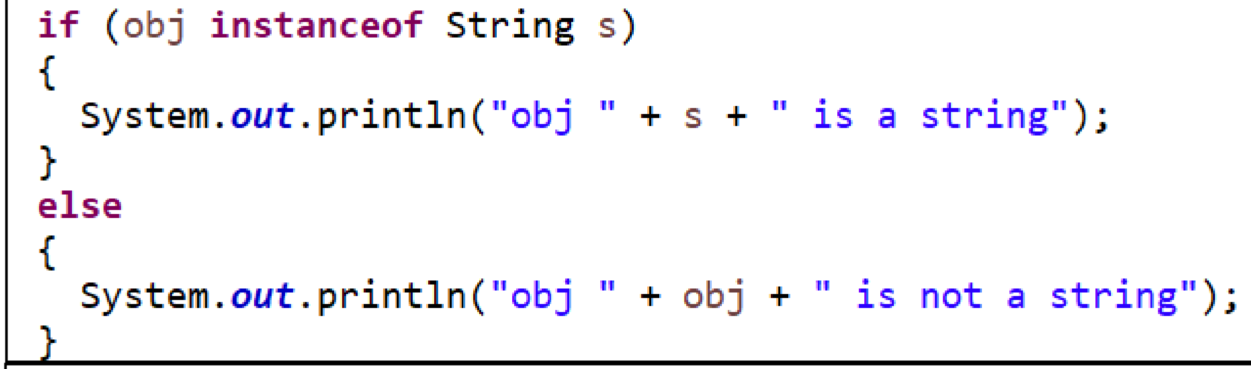
Java SE 14
2nd Preview

Java SE 15
Permanent

Instanceof Pattern Matching

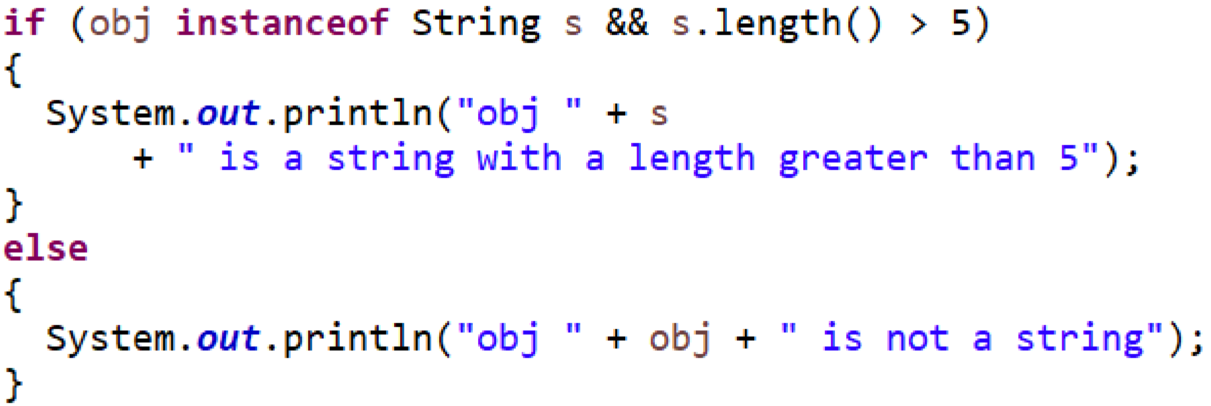


```java
final Object obj = "String";

if (obj instanceof String)
{
  String s = (String) obj;

  System.out.println("obj " + s + " is a string");
}
else
{
  System.out.println("obj " + obj + " is not a string");
}
```



```java
if (obj instanceof String s)
{
  System.out.println("obj " + s + " is a string");
}
else
{
  System.out.println("obj " + obj + " is not a string");
}
```

Java SE 14
1st Preview



```java
if (obj instanceof String s && s.length() > 5)
{
  System.out.println("obj " + s
      + " is a string with a length greater than 5");
}
else
{
  System.out.println("obj " + obj + " is not a string");
}
```

Java SE 15
2nd Preview

Java SE 16
Permanent

Classes & Records

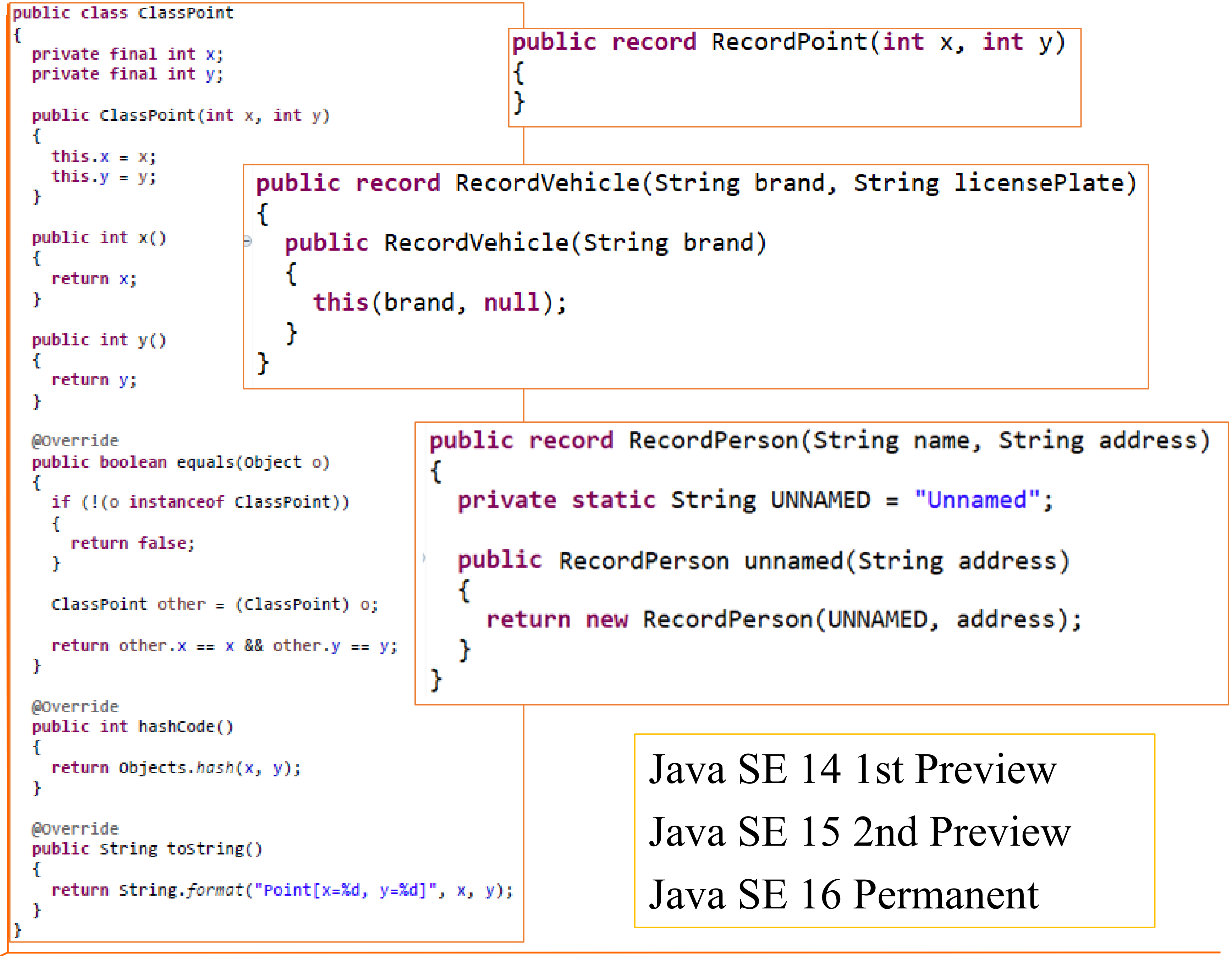
- Passing **data** between objects is one of the most **common**, but **mundane tasks** in many Java applications. In the initial implementation of Java, it requires the creation of a class with **fields** and **methods**, which were susceptible to trivial **mistakes** and muddled intentions
- In many cases, this data is **immutable**, since immutability ensures the **validity** of the data **without synchronization**
- While **IDEs** can automatically **generate** many of these **classes**, they **fail** to automatically **update** classes because, for example, they are not able to **update** the **equals** method to incorporate a new field
- **Records** are **immutable data** classes that require only the **type** and **name** of **fields.** The necessary **methods**, the final fields, and public constructor, are **generated** by the **compiler**

Record Automatic Members and Constraints

- A **private final field** for each component of the state description
- A **public read accessor** method for **each component** of the state description, with the same name and type as the component
- A **public constructor**, whose **signature** is the same as the state **description**, which initializes each field from the corresponding argument
- Implementations of **equals** and **hashCode** methods that say two records are equal if they are of the same type and contain the same state
- An implementation of **toString** method that includes the string representation of all the record components, with their names

- Records **cannot extend** any other **class**, and **cannot declare instance fields** other than the private final fields which correspond to components of the state description. Any **other fields** which are declared must be **static**. These restrictions ensure that the state description alone defines the representation. Records can contains some **additional methods** and implement interfaces
- Records **are** implicitly **final**, and **cannot** be abstract. These restrictions emphasize that the API of a record is defined solely by its state description, and cannot be enhanced later by another class or record.
- Records can **implement interfaces**

Class & Record Code



```java
public class ClassPoint
{
  private final int x;
  private final int y;

  public ClassPoint(int x, int y)
  {
    this.x = x;
    this.y = y;
  }

  public int x()
  {
    return x;
  }

  public int y()
  {
    return y;
  }

  @Override
  public boolean equals(Object o)
  {
    if (!(o instanceof ClassPoint))
    {
      return false;
    }

    ClassPoint other = (ClassPoint) o;

    return other.x == x && other.y == y;
  }

  @Override
  public int hashCode()
  {
    return Objects.hash(x, y);
  }

  @Override
  public String toString()
  {
    return String.format("Point[x=%d, y=%d]", x, y);
  }
}
```

```java
public record RecordPoint(int x, int y)
{
}
```

```java
public record RecordVehicle(String brand, String licensePlate)
{
  public RecordVehicle(String brand)
  {
    this(brand, null);
  }
}
```

```java
public record RecordPerson(String name, String address)
{
  private static String UNNAMED = "Unnamed";

  public RecordPerson unnamed(String address)
  {
    return new RecordPerson(UNNAMED, address);
  }
}
```

Java SE 14 1st Preview

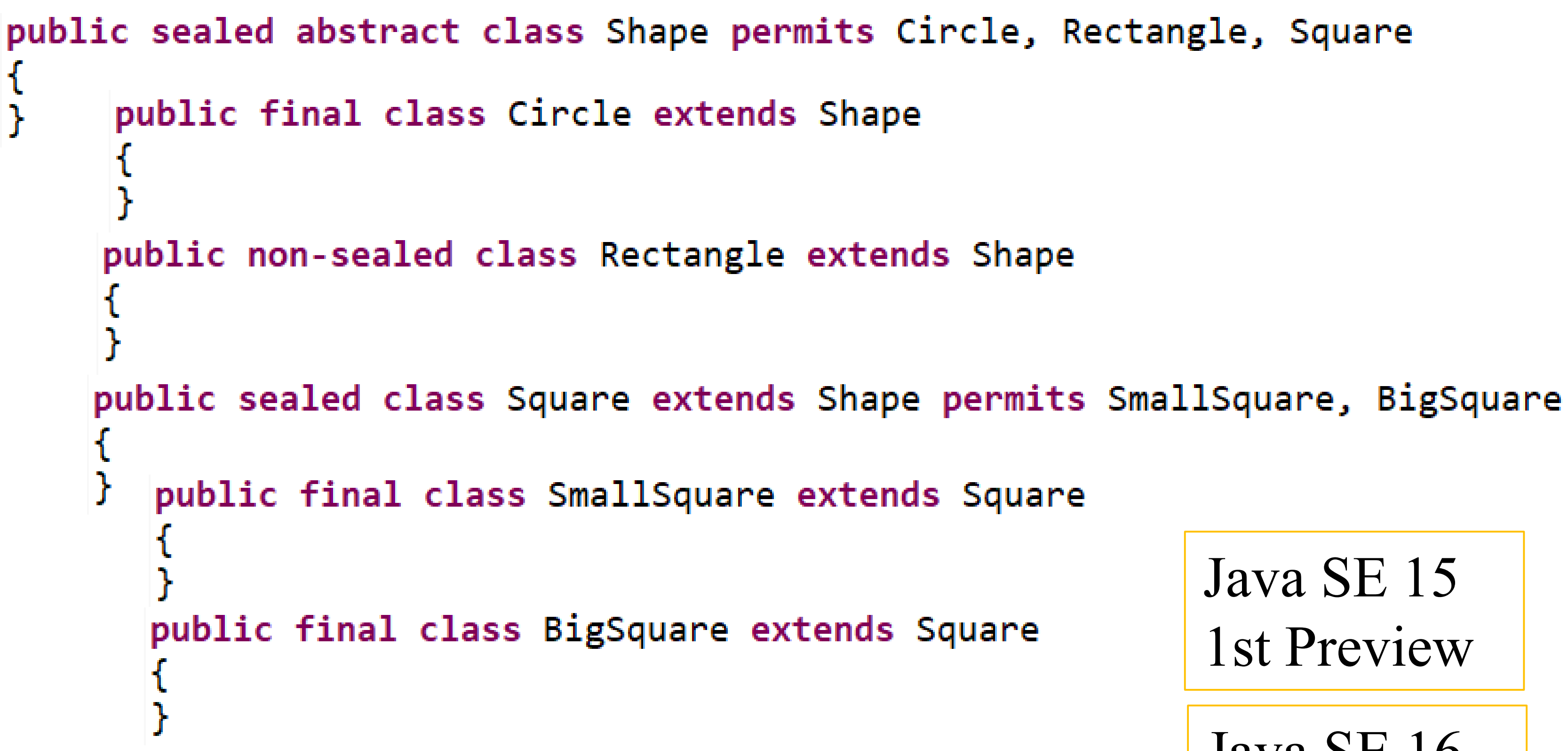
Java SE 15 2nd Preview

Java SE 16 Permanent

Sealed Classes

- A **class hierarchy** enables us to **reuse code** via **inheritance**. However, the **class hierarchy** can also have **other purposes**. Code reuse is great but is not always our primary goal
- An alternative purpose of a **class hierarchy** can be to **model** various possibilities that exist in a **domain**
- As an example, imagine a **business domain** that only works with **cars** and **trucks**, **not motorcycles**
- When creating the **Vehicle abstract class** in Java, we should **be able** to **allow** only **Car** and **Truck classes** to extend it. In that way, we want to ensure that there will be no misuse of the Vehicle abstract class within our domain
- This feature is about enabling more **fine-grained** inheritance **control** in Java. **Sealing** allows **classes** and **interfaces** to define their **permitted subtypes**
- In other words, a **class** or an interface can now **define** which **classes** can **implement** or **extend** it. It is a **useful** feature for **domain modeling** and increasing the **security** of **libraries**

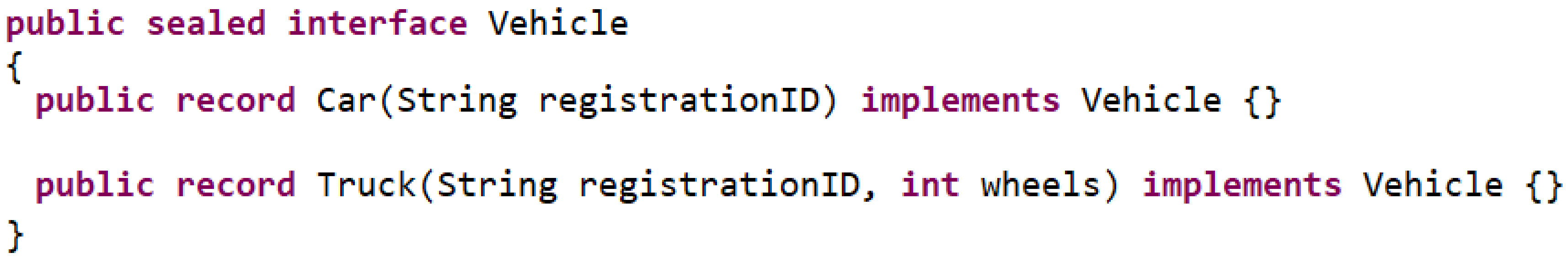
Sealed Class and Interface Code



```
public sealed abstract class Shape permits Circle, Rectangle, Square
{
}     public final class Circle extends Shape
      {
      }
     public non-sealed class Rectangle extends Shape
     {
     }
    public sealed class Square extends Shape permits SmallSquare, BigSquare
    {
    }  public final class SmallSquare extends Square
       {
       }
       public final class BigSquare extends Square
       {
       }
```

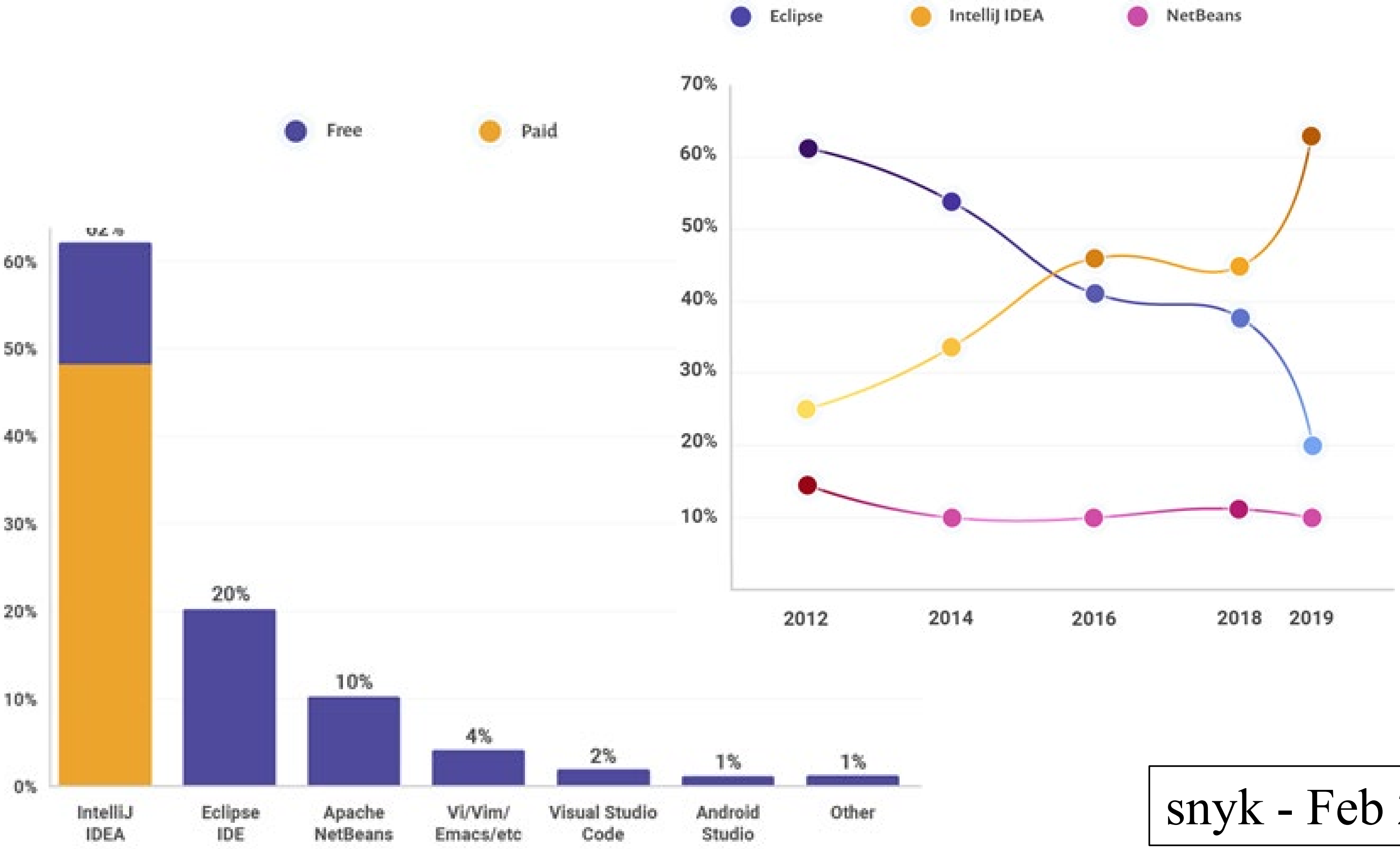
Java SE 15
1st Preview

Java SE 16
2nd Preview



```
public sealed interface Vehicle
{
  public record Car(String registrationID) implements Vehicle {}

  public record Truck(String registrationID, int wheels) implements Vehicle {}
}
```

IDE (1/2)



snyk - Feb 2020

IDE (2/2)

- OpenJDK 16 (March 16, 2021)
  - http://jdk.java.net/16/
- Eclipse 2021-03 (March 17, 2021)
  - https://www.eclipse.org/downloads/
  - Two recommended configurations, i.e., Java Developers and Java Enterprise Developers
  - https://marketplace.eclipse.org/content/java-16-support-eclipse-2021-03-419
  - Update for JDK 16
- IntellyJ IDEA
  - https://www.jetbrains.com/idea/
- Apache NetBeans
  - https://netbeans.apache.org/download/