being selected at runtime. These methods often leverage generic types, promoting code reusability and flexibility. Furthermore, *VarHandles*, as an example of a signature polymorphic method, can handle multiple call sites and support various return types.

The variable access modes can be classified as follows based on their type:

1. Read access (e.g., `getVolatile`): For reading variables with *volatile* memory ordering effects.
2. Write access (e.g., `setVolatile`): For updating variables with *release* memory ordering effects.
3. Read–modify–write access (e.g., `compareAndExchange`, `getAndAddRelease`, `getAndBitwiseXorRelease`): Encompasses three types:
    a. Atomic update: For performing *compare-and-set* operations on variables with *volatile* memory ordering.
    b. Numeric atomic update: For performing *get-and-add* with *release* memory-order effects for updating and *acquire* memory-order effects for reading.
    c. Bitwise atomic update: For performing *get-and-bitwise-xor* with *release* memory-order effects for updating and *volatile* memory-order effects for reading.

The goals of the JDK Enhancement Proposal (JEP) for variable handles[2] were to have predictable performance (as it doesn't involve boxing or arguments packing) equivalent to what `sun.misc.Unsafe` provides, safety (no corrupt state caused by access or update), usability (superior to `sun.misc.Unsafe`), and integrity similar to that provided by `getfield`, `putfield`, and non-updating final fields.

Let's explore an example that demonstrates the creation of a *VarHandle* instance for a static field. Note that we will follow the conventions outlined in "Using JDK 9 Memory Order Modes";[3] that is, *VarHandles* will be declared as static final fields and initialized in static blocks. Also, *VarHandle* field names will be in uppercase.

```
// Declaration and Initialization Example
class SafePin {
    int pin;
}

class ModifySafePin {
    // Declare a static final VarHandle for the "pin" field
    private static final VarHandle VH_PIN;

    static {
         try {
            // Initialize the VarHandle by finding it for the "pin" field of SafePin class
            VH_PIN = MethodHandles.lookup().findVarHandle(SafePin.class, "pin", int.class);
         } catch (Exception e) {
```

[2] JEP 193: *Variable Handles*. https://openjdk.org/jeps/193.

[3] Doug Lea. "Using JDK 9 Memory Order Modes." https://gee.cs.oswego.edu/dl/html/j9mm.html.

```
            // Throw an error if VarHandle initialization fails
            throw new Error(e);
        }
    }
}
```

In this example, we have successfully created a *VarHandle* for direct access to the `pin` field. This *VarHandle* enables us to perform a bitwise or a numeric atomic update on it.

Now, let's explore an example that demonstrates how to use a *VarHandle* to perform an atomic update:

```
// Atomic Update Example
class ModifySafePin {
    private static final VarHandle VH_PIN = ModifySafePin.getVarHandle();

    private static VarHandle getVarHandle() {
          try {
              // Find and return the VarHandle for the "pin" field of SafePin class
              return MethodHandles.lookup().findVarHandle(SafePin.class, "pin", int.class);
          } catch (Exception e) {
              // Throw an error if VarHandle initialization fails
              throw new Error(e);
          }
    }

    public static void atomicUpdatePin(SafePin safePin, int newValue) {
         int prevValue;
         do {
              prevValue = (int) VH_PIN.getVolatile(safePin);
         } while (!VH_PIN.compareAndSet(safePin, prevValue, newValue));
    }
}
```

In this example, the `atomicUpdatePin` method performs an atomic update on the `pin` field of a `SafePin` object by using the *VarHandle* `VH_PIN`. It uses a loop to repeatedly get the current value of `pin` and attempts to set it to `newValue`. The `compareAndSet` method atomically sets the value of `pin` to `newValue` if the current value is equal to `prevValue`. If another thread modifies the value of `pin` between the read operation (`getVolatile`) and the compare-and-set operation (`compareAndSet`), the `compareAndSet` fails, and the loop will repeat until the update succeeds.

This is a simple example of how to use *VarHandles* to perform atomic updates. Depending on your requirements, you might need to handle exceptions differently, consider synchronization, or utilize other methods provided by *VarHandle*. You should always exercise caution when using *VarHandles* because incorrect usage can lead to data races and other concurrency issues. Thoroughly test your code and consider using higher-level concurrency utilities, when possible, to ensure proper management of concurrent variable access.

Java's Type System Evolution: Java 11 to Java 17

Switch Expressions

Java 12 introduced a significant enhancement to the traditional `switch` statement with the concept of switch expressions. This feature was further refined and finalized in Java 14. Switch expressions simplify the coding patterns for handling multiple cases by automatically inferring the type of the returned value. The `yield` keyword (introduced in Java 13) is used to return a value from a switch block.

One of the key improvements was the introduction of the arrow syntax (->) for case labels, which replaced the traditional colon syntax (:). This syntax not only makes the code more readable but also eliminates the need for `break` statements, reducing the risk of fall-through cases.

Here's an example showing how switch expressions can be used:

```
public enum Day {
   MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}

public String getDayName(Day day) {
   return switch (day) {
      case MONDAY -> "Monday";
      case TUESDAY -> "Tuesday";
      case WEDNESDAY, THURSDAY -> {
         String s = day == Day.WEDNESDAY ? "Midweek" : "Almost Weekend";
         yield s; // yield is used to return a value from a block of code
      }
      // Other cases …
      default -> throw new IllegalArgumentException("Invalid day: " + day);
   };
}
```

In this example, the switch expression is used to return the name of a day based on the value of the `day` variable. The `yield` keyword is used to return a value from a block of code in the case of `WEDNESDAY` and `THURSDAY`.

Switch expressions also improve the compiler's exhaustiveness checking. As a result, the compiler can detect whether all possible cases are covered in a switch expression and issue a warning if any are missing. This feature is especially beneficial when you are dealing with `enum` values. In a traditional `switch` statement, if a new `enum` value is introduced, it could easily be overlooked in the corresponding switch cases, leading to potential bugs or unexpected behavior. However, with switch expressions, the compiler will prompt you to explicitly handle these new cases. This not only ensures code correctness but also enhances maintainability by making it easier to accommodate later changes in your `enum` definitions.

From a maintainability perspective, switch expressions can significantly improve the readability of your code. The compiler can optimize the execution of switch expressions effectively as is

possible with traditional `switch` statements. This optimization is particularly noticeable when you are dealing with cases involving strings.

In traditional `switch` statements, handling string cases involves a sequence of operations by the Java compiler, which may include techniques such as hashing and the use of lookup tables as a part of the compiled bytecode to efficiently manage to jump to the correct case label. This process may seem less direct, but the compiler is equipped to optimize these operations for a multitude of cases.

Switch expressions allow for more concise code and eliminate the possibility of fall-through, which can simplify the compiler's task when generating efficient bytecode. This is particularly helpful with multiple case labels, enabling a streamlined way to handle cases that yield the same result. This approach drastically reduces the number of comparisons needed to find the correct case, thereby significantly streamlining code complexity when there are a substantial number of cases.

Sealed Classes

Sealed classes were introduced as a preview feature in JDK 15 and finalized in JDK 17. They provide developers with a way to define a restricted class hierarchy. This feature allows for better control over class inheritance, thereby simplifying code maintenance and comprehension.

Here's an example that illustrates the concept of sealed classes using various types of pets:

```
sealed abstract class Pet permits Mammal, Reptile {}

sealed abstract class Mammal extends Pet permits Cat, Dog, PygmyPossum {
   abstract boolean isAdult();
}

final class Cat extends Mammal {
    boolean isAdult() {
      return true;
    }
    // Cat-specific properties and methods
}

// Similar implementations for Dog and PygmyPossum

sealed abstract class Reptile extends Pet permits Snake, BeardedDragon {
    abstract boolean isHarmless();
}

final class Snake extends Reptile {
    boolean isHarmless() {
        return true;
    }
    // Snake-specific properties and methods
}

// Similar implementation for BeardedDragon
```

In this example, we have a sealed class `Pet` that permits two subclasses: `Mammal` and `Reptile`. Both of these subclasses are also sealed: `Mammal` permits `Cat`, `Dog`, and `PygmyPossum`, while `Reptile` permits `Snake` and `BeardedDragon`. Each of these classes has an `isAdult` or `isHarmless` method, indicating whether the pet is an adult or is harmless, respectively. By organizing the classes in this way, we establish a clear and logical hierarchy representing different categories of pets.

This well-structured hierarchy demonstrates the power and flexibility of sealed classes in Java, which enable developers to define restricted and logically organized class hierarchies. The JVM fully supports sealed classes and can enforce the class hierarchy defined by the developer at the runtime level, leading to robust and error-free code.

Records

Java 16 introduced records as a preview feature; they became a standard feature in Java 17. Records provide a compact syntax for declaring classes that are primarily meant to encapsulate data. They represent data as a set of properties or fields and automatically generate related methods such as accessors and constructors. By default, records are immutable, which means that their state can't be changed post creation.

Let's look at an example of `Pet` records:

```
List<Pet> pets = List.of(
    new Pet("Ruby", 2, "Great Pyrenees and Red Heeler Mix", true),
    new Pet("Bash", 1, "Maltese Mix", false),
    new Pet("Perl", 10, "Black Lab Mix", true)
);

List<Pet> adultPets = pets.stream()
    .filter(Pet::isAdult)
    .collect(Collectors.toList());

adultPets.forEach(pet -> System.out.println(pet.name() + " is an adult " + pet.breed()));
```

In this code, we create a list of `Pet` records and then use a stream to filter out the pets that are adults. The `filter` method uses the `Pet::isAdult` method reference to filter the stream. The `collect` method then gathers the remaining elements of the stream into a new list. Finally, we use a `forEach` loop to print out the names and breeds of the adult pets.

Incorporating records into your code can enhance its clarity and readability. By defining the components of your class as part of the record definition, you can avoid having to write boilerplate code for constructors, accessor methods, and `equals` and `hashCode` implementations. Additionally, because records are immutable by default, they can be safely shared between threads without the need for synchronization, which can further improve performance in multithreaded environments.

Beyond Java 17: Project Valhalla

Project Valhalla is a highly anticipated enhancement to Java's type system. It aims to revolutionize the way we understand and use the Java object model and generics. The main goals are to introduce inline classes (also known as value classes or types) and user-defined primitives. The project also aims to make generics more versatile to support these new types. These changes are expected to lead to improved performance characteristics, particularly for small, immutable objects and generic APIs. To fully appreciate the potential of these changes, we need to take a closer look at the current performance implications of Java's type system.

Performance Implications of the Current Type System

Java's type system distinguishes between primitive data types and reference types (objects). Primitive data types don't possess an identity; they represent pure data values. The JVM assigns identity only to aggregate objects, such as classes and arrays, such that each instance is distinct and can be independently tracked by the JVM.

This distinction has notable performance implications. Identity allows us to mutate the state of an object, but it comes with a cost. When considering an object's state, we need to account for the object header size and the locality implications of the pointers to this object. Even for an immutable object, we still need to account for its identity and provide support for operations like `System.identityHashCode`, as the JVM needs to be prepared for potential synchronization with respect to the object.

To better understand this, let's visualize a Java object (Figure 2.2). It consists of a header and a body of fields or, in the case of an array object, array elements. The object header includes a mark word and a `klass`, which points to the class's metadata (Figure 2.3). For arrays, the header also contains a length word.

Figure 2.2 A Java Object

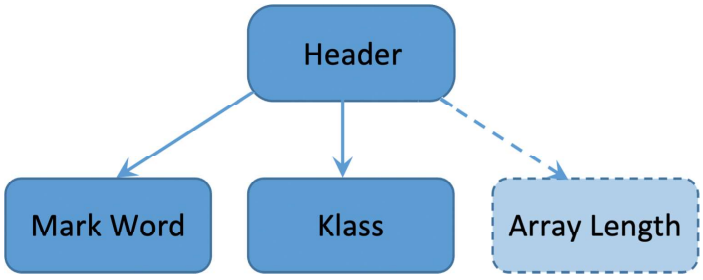

Figure 2.3 A Java Object's Header

NOTE The `klass` will be compressed if `–XX:+UseCompressedOops` is enabled.

Analyzing Object Memory Layout with Java Object Layout (JOL)

Java Object Layout (JOL)[4] is a powerful tool for understanding the memory allocation of objects. It provides a clear breakdown of the overhead associated with each object, including the object header and fields. Here's how we can analyze the layout of an 8-byte array using JOL:

```
import org.openjdk.jol.info.ClassLayout;
import org.openjdk.jol.vm.VM;
import static java.lang.System.out;

public class ArrayLayout {
    public static void main(String[] args) throws Exception {
        out.println(VM.current().details());
        byte[] ba = new byte[8];
        out.println(ClassLayout.parseInstance(ba).toPrintable());
    }
}
```

For more information on how to print internal details, refer to the following two `jol-core` files:[5]

```
org/openjdk/jol/info/ClassLayout.java
org/openjdk/jol/info/ClassData.java
```

For simplicity, let's run this code on 64-bit hardware with a 64-bit operating system and a 64-bit Java VM with `–XX:+UseCompressedOops` enabled. (This option has been enabled by default since Java 6, update 23. If you are unsure which options are enabled by default, use the `–XX:+PrintFlagsFinal` option to check.) Here's the output from JOL:

```
# Running 64-bit HotSpot VM.
# Using compressed oop with 3-bit shift.
# Using compressed klass with 3-bit shift.
# WARNING | Compressed references base/shifts are guessed by the experiment!
# WARNING | Therefore, computed addresses are just guesses, and ARE NOT RELIABLE.
# WARNING | Make sure to attach Serviceability Agent to get the reliable addresses.
# Objects are 8 bytes aligned.
# Field sizes by type: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]
# Array element sizes: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]
```

[4] OpenJDK. "Code Tools: jol." https://openjdk.org/projects/code-tools/jol/.

[5] Maven Repository. "Java Object Layout: Core." https://mvnrepository.com/artifact/org.openjdk.jol/jol-core.

```
[B object internals:
OFFSET SIZE TYPE DESCRIPTION     VALUE
   0     4       (object header) 01 00 00 00 (00000001 …) (1)
   4     4       (object header) 00 00 00 00 (00000000 …) (0)
   8     4       (object header) 48 68 00 00 (01001000 …) (26696)
  12     4       (object header) 08 00 00 00 (00001000 …) (8)
  16     8 byte  [B.<elements>          N/A
Instance size: 24 bytes
Space losses: 0 bytes internal + 0 bytes external = 0 bytes total
```

Breaking down the output, we can see several sections: the JVM configuration, the byte array internals, the mark word, the `klass`, the array length, the array elements, and possible padding for alignment.

Now let's parse the output in sections.

Section 1: The first three lines

```
# Running 64-bit HotSpot VM.
# Using compressed oop with 3-bit shift.
# Using compressed klass with 3-bit shift.
```

These lines confirm that we are on a 64-bit Java VM with compressed ordinary object pointers (oops) enabled, and the `klass` is also compressed. The `klass` pointer is a critical component of the JVM as it accesses metadata for the object's class, such as its methods, fields, and superclass. This metadata is necessary for many operations, such as method invocation and field access.

Compressed oops[6] and `klass` pointers are employed to reduce memory usage on 64-bit JVMs. Their use can significantly improve performance for applications that have a large number of objects or classes.

Section 2: Byte array internals

```
[B object internals:
OFFSET SIZE TYPE DESCRIPTION     VALUE
   0     4       (object header) 01 00 00 00 (00000001 …) (1)
   4     4       (object header) 00 00 00 00 (00000000 …) (0)
```

The output also shows the internals of a byte array. Referring back to Figure 2.3, which shows the Java object's header, we can tell that the two lines after the title represent the mark word. The mark word is used by the JVM for object synchronization and garbage collection.

Section 3: `klass`

```
OFFSET SIZE TYPE DESCRIPTION     VALUE
   8     4       (object header) 48 68 00 00 (01001000 …) (26696)
```

[6] https://wiki.openjdk.org/display/HotSpot/CompressedOops

The `klass` section shows only 4 bytes because when compressed oops are enabled, the compressed `klass` is automatically enabled as well. You can try disabling compressed `klass` by using the following option: `-XX:-UseCompressedClassPointers`.

Section 4: Array length

```
OFFSET SIZE TYPE DESCRIPTION     VALUE
  12     4       (object header) 08 00 00 00 (00001000 …) (8)
```

The array length section shows a value of 8, indicating this is an 8-byte array. This value is the array length field of the header.

Section 5: Array elements

```
OFFSET SIZE TYPE DESCRIPTION     VALUE
  16     8 byte  [B.<elements>   N/A
```

The array elements section displays the body/field `array.elements`. Since this is an 8-byte array, the size is shown as 8 bytes in the clarified output. Array elements are stored in a contiguous block of memory, which can improve cache locality and performance.

Understanding the composition of an array object allows us to appreciate the overhead involved. For arrays with sizes of 1 to 8 bytes, we need 24 bytes (the required padding can range from 0 to 7 bytes, depending on the elements):

```
Instance size: 24 bytes
Space losses: 0 bytes internal + 0 bytes external = 0 bytes total
```

The instance size and space losses are calculated to understand the memory usage of your objects better.

Now, let's examine a more complex example: an array of the `MorningPeople` class. First, we will consider the mutable version of the class:

```
public class MorningPeopleArray {

    public static void main(String[] args) throws Exception {
        out.println(VM.current().details());

        // Create an array of MorningPeople objects
        MorningPeople[] mornpplarray = new MorningPeople[8];

        // Print the layout of the MorningPeople class
        out.println(ClassLayout.parseClass(MorningPeople.class).toPrintable());

        // Print the layout of the mornpplarray instance
```

```
        out.println(ClassLayout.parseInstance(mornpplarray).toPrintable());
    }
}
class MorningPeople {
    String name;
    Boolean type;

    public MorningPeople(String name, Boolean type) {
        this.name = name;
        this.type = type;
    }
}
```

The output would now appear as follows:

```
# Running 64-bit HotSpot VM.
# Using compressed oop with 3-bit shift.
# Using compressed klass with 3-bit shift.
# WARNING | Compressed references base/shifts are guessed by the experiment!
# WARNING | Therefore, computed addresses are just guesses, and ARE NOT RELIABLE.
# WARNING | Make sure to attach Serviceability Agent to get the reliable addresses.
# Objects are 8 bytes aligned.
# Field sizes by type: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]
# Array element sizes: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]

MorningPeople object internals:
 OFFSET  SIZE   TYPE               DESCRIPTION                VALUE
      0    12                      (object header)            N/A
     12     4   java.lang.String   MorningPeople.name         N/A
     16     4   java.lang.Boolean  MorningPeople.type         N/A
     20     4                      (loss due to the next object alignment)
Instance size: 24 bytes
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total

[LMorningPeople; object internals:
 OFFSET  SIZE   TYPE              DESCRIPTION               VALUE
      0     4                     (object header)           (1)
      4     4                     (object header)           (0)
      8     4                     (object header)           (13389376)
     12     4                     (object header)           (8)
     16    32   MorningPeople     MorningPeople;.<elements> N/A
Instance size: 48 bytes
Space losses: 0 bytes internal + 0 bytes external = 0 bytes total
```

Inefficiency with Small, Immutable Objects

Consider the immutable `MorningPeople` class:

```
// An immutable class by declaring it final
final class MorningPeople {
    private final String name;
    private final Boolean type;

    public MorningPeople(String name, Boolean type) {
        this.name = name;
        this.type = type;
    }
    // Getter methods for name and type
    public String getName() {
         return name;
    }
    public Boolean getType() {
        return type;
    }
}
```

In this version of the `MorningPeople` class, the `name` and `type` fields are `private` and `final`, meaning they can't be modified once they're initialized in the constructor. The class is also declared as `final`, so it can't be subclassed. There are no setter methods, so the state of a `MorningPeople` object can't be changed after it's created. This makes the `MorningPeople` class immutable.

If we compile and run this code through JOL, the internals will still be the same as we saw with the mutable version. This outcome highlights an important point: The current Java type system is inefficient for certain types of data. This inefficiency is most evident in cases where we are dealing with large quantities of small, immutable data objects. For such cases, the memory overhead becomes significant, and the issue of locality of reference comes into play. Each `MorningPeople` object is a separate entity in memory, even though the latter is immutable. The references to these objects are scattered across the heap. This scattered layout can, in turn, adversely affect the performance of our Java programs due to poor hardware cache utilization.

The Emergence of Value Classes: Implications for Memory Management

Project Valhalla's planned introduction of value classes is poised to transform the JVM's approach to memory management. By reducing the need for object identity in certain cases, these classes are expected to lighten the garbage collector's workload, leading to fewer pauses and more efficient GC cycles. This change also aims to optimize storage and retrieval operations, especially for arrays, by leveraging the compact layout of value types. However, the path

to integrating these new classes poses challenges, particularly in managing the coexistence of value and reference types. The JVM must adapt its GC strategies and optimize cache usage to handle this new mixed model effectively.

Value classes represent a fundamental shift from traditional class structures. They lack identity, so they differ from objects in the conventional sense. They are proposed to be immutable, which would mean they cannot be synchronized or mutated post creation—attributes that are critical to maintaining state consistency. In the current Java type system, to create an immutable class, as in the `MorningPeople` example, you would typically declare the class as `final` (preventing extension), make all fields `private` (disallowing direct access), omit setter methods, make all mutable fields `final` (allowing only one-time assignment), initialize all fields via a constructor, and ensure exclusive access to any mutable components.

Value classes envision a streamlined process and promise a more memory-efficient storage method by eliminating the need for an object header, enabling inline storage akin to primitive types. This results in several key benefits:

- **Reduced memory footprint:** Inline storage significantly reduces the required memory footprint.
- **Improved locality of reference:** Inline storage ensures that all the fields of a value class are located close to each other in memory. This improved locality of reference can significantly enhance the performance of our programs by optimizing the CPU cache usage.
- **Versatile field types:** Value classes are predicted to support fields of any type, including primitives, references, and nested value classes. This flexibility to have user-defined primitive types allows for more expressive and efficient code.

Redefining Generics with Primitive Support

Recall our discussion of generics in Chapter 1, where we examined the generic type `FreshmenAdmissions<K, V>` using `String` and `Boolean` as type parameters for `K` and `V`, respectively. However, in current Java, generics are limited to reference types. Primitive types, if needed, are used via autoboxing with their wrapper classes, as demonstrated in the snippet from Chapter 1:

```
applicationStatus.admissionInformation("Monica", true);
```

Under Project Valhalla, the evolution of Java's type system aims to include support for generics over primitive types, eventually extending this support to value types. Consider the potential definition of the `FreshmenAdmissions` class in this new context:

```
public class FreshmenAdmissions<K, any V> {
    K key;
    V boolornumvalue;

    public void admissionInformation(K name, V value) {
```

```
        key = name;
        boolornumvalue = value;
    }
}
```

Here, the `any` keyword introduces a significant shift, allowing generics to directly accept primitive types without the need for boxing. For instance, in the modified `FreshmenAdmissions<K,any V>` class, `V` can now directly be a primitive type.[7] To demonstrate this enhancement, consider the `TestGenerics` class using the primitive type `boolean` instead of the wrapper class `Boolean`:

```
public class TestGenerics {
    public static void main(String[] args) {
        FreshmenAdmissions<String, boolean> applicationStatus = new FreshmenAdmissions<>();
        applicationStatus.admissionInformation("Monica", true);
    }
}
```

This example illustrates how the direct use of primitive types in generics could simplify code and improve performance, a key goal of Project Valhalla's advancements in the Java type system.

Exploring the Current State of Project Valhalla

Classes and Types

Project Valhalla introduces several novel concepts to the Java class hierarchy, such as inline (value) classes, primitive classes, and enhanced generics. Value classes, which were discussed in the preceding section, have a unique characteristic: When two instances of a value class are compared using the == operator, it checks for content equality rather than instance identity (because value classes do not have identity). Furthermore, synchronization on value classes is not possible. The JVM can use these features to save memory by avoiding unnecessary heap allocations.

Primitive classes are a special type of value class. They're not reference types, which means they can't be null. Instead, if you don't assign them a value, their values will default to zero. Primitive classes are stored directly in memory, rather than requiring a separate heap allocation. As a result, arrays of primitive classes don't need to point to different parts of the heap for each element, which makes programs run both more quickly and more efficiently. When necessary, the JVM can "box" a primitive class into a value class, and vice versa.[8]

[7] Nicolai Parlog. "Java's Evolution into 2022: The State of the Four Big Initiatives." Java Magazine, February 18, 2022. https://blogs.oracle.com/javamagazine/post/java-jdk-18-evolution-valhalla-panama-loom-amber.

[8] JEP 401: *Value Classes and Objects (Preview).* https://openjdk.org/jeps/401.

Significantly, Project Valhalla also introduces primitive classes for the eight basic types in the JVM: `byte`, `char`, `short`, `int`, `long`, `boolean`, `float`, and `double`. This makes the type system more elegant and easier to understand because it is no longer necessary to treat these types differently from other types. The introduction of these primitive classes for the basic types is a key aspect of Project Valhalla's efforts to optimize Java's performance and memory management, making the language more efficient for a wide range of applications.

Project Valhalla is working on improving generics so that they will work better with value classes. The goal is to help generic APIs perform better when used with value classes.

Memory Access Performance and Efficiency

One of the main goals of Project Valhalla is to improve memory access performance and efficiency. By allowing value classes and primitive classes to be stored directly in memory, and by improving generics so that they work better with these types, Project Valhalla aims to reduce memory usage and improve the performance of Java applications. This would be especially beneficial for lists or arrays, which can store their values directly in a block of memory, without needing separate heap pointers.

NOTE To stay updated on the latest advancements and track the ongoing progress of Project Valhalla, you have several resources at your disposal. The OpenJDK's Project Valhalla mailing list[9] is a great starting point. Additionally, you can export the GitHub repository[10] for in-depth insights and access to source codes. For hands-on experience, the latest early-access downloadable binaries[11] offer a practical way to experiment with OpenJDK prototypes, allowing you to explore many of these new features.

Early Access Release: Advancing Project Valhalla's Concepts

The early access release of Project Valhalla JEP 401: *Value Classes and Objects*[12] marks a significant step in realizing the project's ambitious goals. While previous sections discussed the theoretical framework of value classes, the early access (EA) release concretizes these concepts with specific implementations and functionalities. Key highlights include

- **Implementation of value objects:** In line with the Java Language Specification for Value Objects,[13] this release introduces value objects in Java, emphasizing their identity-free behavior and the ability to have inline, allocation-free encodings in local variables or method arguments.

[9] https://mail.openjdk.org/mailman/listinfo/valhalla-dev

[10] https://github.com/openjdk/valhalla/tree/lworld

[11] https://jdk.java.net/valhalla/

[12] https://openjdk.org/jeps/401

[13] https://cr.openjdk.org/~dlsmith/jep8277163/jep8277163-20220830/specs/value-objects-jls.html#jls-8.1.1.5

- **Enhancements and experimental features:** The EA release experiments with features like flattened fields and arrays, and developers can use the `.ref` suffix to refer to the corresponding reference type of a primitive class. This approach aims to provide null-free primitive value types, potentially transforming Java's approach to memory management and data representation.
- **Command-line options for experimentation:** Developers can use options like `-XDenablePrimitiveClasses` and `-XX:+EnablePrimitiveClasses` to unlock experimental support for *Primitive Classes*, while other options like `–XX:InlineFieldMaxFlatSize=n` and `–XX:FlatArrayElementMaxSize=n` allow developers to set limits on flattened fields and array components.
- **HotSpot optimizations:** Significant optimizations in HotSpot's C2 compiler specifically target the efficient handling of value objects, aiming to reduce the need for heap allocations.

With these enhancements, Project Valhalla steps out of the theoretical realm and into a more tangible form, showcasing the practical implications and potential of its features. As developers explore these features, it's important to note the compatibility implications. Refactoring an identity class to a value class may affect existing code. Additionally, value classes can now be declared as records and made serializable, expanding their functionality. Core reflection has also been updated to support these new modifiers. Finally, it's crucial for developers to understand that value class files generated by *javac* in this EA release are specifically tailored for this version. They may not be compatible with other JVMs or third-party tools, indicating the experimental nature of these features.

Use Case Scenarios: Bringing Theory to Practice

In high-performance computing and data processing applications, value classes can offer substantial benefits. For example, financial simulations that handle large volumes of immutable objects can leverage the memory efficiency and speed of value classes for more efficient processing.[14] Enhanced generics enable API designers to create more versatile yet less complex APIs, promoting broader applicability and easier maintenance.

Concurrent applications stand to benefit significantly from the inherent thread safety of value classes. The immutability of value classes aligns with thread safety, reducing the complexity associated with synchronization in multithreaded environments.

A Comparative Glance at Other Languages

Java's upcoming value classes, part of Project Valhalla, aim to optimize memory usage and performance similar to C#'s value types. However, Java's value classes are expected to offer enhanced features for methods and interfaces, diverging from C#'s more traditional approach with structs. While C#'s structs provide efficient memory management, they face performance costs associated with boxing when used as reference types. Java's approach is focused

[14] https://openjdk.java.net/projects/valhalla/

on avoiding such overhead and enhancing runtime efficiency, particularly in collections and arrays.[15]

Kotlin's data classes offer functional conveniences akin to Java's records, automating common methods like `equals()` and `hashCode()`, but they aren't tailored for memory optimization. Instead, Kotlin's inline classes are the choice for optimizing memory. Inline classes avoid overhead by embedding values directly in the code during compilation, which is comparable to the goals of Java's proposed value classes in Project Valhalla for reducing runtime costs.[16]

Scala's case classes are highly effective for representing immutable data structures, promoting functional programming practices. They inherently provide pattern matching, `equals()`, `hashCode()`, and `copy()` methods. However, unlike Java's anticipated value classes, Scala's case classes do not specialize in memory optimization or inline storage. Their strength lies in facilitating immutable, pattern-friendly data modeling, rather than in low-level memory management or performance optimization typical of value types.[17]

Conclusion

In this chapter, we've explored the significant strides made in Java's type system, leading up to the advancements now offered by Project Valhalla. From the foundational use of primitive and reference types to the ambitious introduction of value types and enhanced generics, Java's evolution reflects a consistent drive toward efficiency and performance.

Project Valhalla, in particular, marks a notable point in this journey, bringing forward concepts that promise to refine Java's capabilities further. These changes are pivotal for both new and experienced Java developers, offering tools for more efficient and effective coding. Looking ahead, Project Valhalla sets the stage for Java's continued relevance in the programming world—it represents a commitment to modernizing the language while maintaining its core strengths.

[15] https://docs.microsoft.com/en-us/dotnet/csharp/

[16] https://kotlinlang.org/docs/reference/data-classes.html

[17] https://docs.scala-lang.org/tour/case-classes.html

Chapter 3

From Monolithic to Modular Java: A Retrospective and Ongoing Evolution

Introduction

In the preceding chapters, we journeyed through the significant advancements in the Java language and its execution environment, witnessing the remarkable growth and transformation of these foundational elements. However, a critical aspect of Java's evolution, which has far-reaching implications for the entire ecosystem, is the transformation of the Java Development Kit (JDK) itself. As Java matured, it introduced a plethora of features and language-level enhancements, each contributing to the increased complexity and sophistication of the JDK. For instance, the introduction of the enumeration type in J2SE 5.0 necessitated the addition of the `java.lang.Enum` base class, the `java.lang.Class.getEnumConstants()` method, `EnumSet`, and `EnumMap` to the `java.util` package, along with updates to the *Serialized Form*. Each new feature or syntax addition required meticulous integration and robust support to ensure seamless functionality.

With every expansion of Java, the JDK began to exhibit signs of unwieldiness. Its monolithic structure presented challenges such as an increased memory footprint, slower start-up times, and difficulties in maintenance and updates. The release of JDK 9 marked a significant turning point in Java's history, as it introduced the Java Platform Module System (JPMS) and transitioned Java from a monolithic structure to a more manageable, modular one. This evolution continued with JDK 11 and JDK 17, with each bringing further enhancements and refinements to the modular Java ecosystem.

This chapter delves into the specifics of this transformation. We will explore the inherent challenges of the monolithic JDK and detail the journey toward modularization. Our discussion will extend to the benefits of modularization for developers, particularly focusing on those who have adopted JDK 11 or JDK 17. Furthermore, we'll consider the impact of these changes on JVM performance engineering, offering insights to help developers optimize their applications

and leverage the latest JDK innovations. Through this exploration, the goal is to demonstrate how Java applications can significantly benefit from modularization.

Understanding the Java Platform Module System

As just mentioned, the JPMS was a strategic response to the mounting complexity and unwieldiness of the monolithic JDK. The primary goal when developing it was to create a scalable platform that could effectively manage security risks at the API level while enhancing performance. The advent of modularity within the Java ecosystem empowered developers with the flexibility to select and scale modules based on the specific needs of their applications. This transformation allowed Java platform developers to use a more modular layout when managing the Java APIs, thereby fostering a system that was not only more maintainable but also more efficient. A significant advantage of this modular approach is that developers can utilize only those parts of the JDK that are necessary for their applications; this selective usage reduces the size of their applications and improves load times, leading to more efficient and performant applications.

Demystifying Modules

In Java, a module is a cohesive unit comprising packages, resources, and a module descriptor (`module-info.java`) that provides information about the module. The module serves as a container for these elements. Thus, a module

- **Encapsulates its packages:** A module can declare which of its packages should be accessible to other modules and which should be hidden. This encapsulation improves code maintainability and security by allowing developers to clearly express their code's intended usage.
- **Expresses dependencies:** A module can declare dependencies on other modules, making it clear which modules are required for that module to function correctly. This explicit dependency management simplifies the deployment process and helps developers identify problematic issues early in the development cycle.
- **Enforces strong encapsulation:** The module system enforces strong encapsulation at both compile time and runtime, making it difficult to break the encapsulation either accidentally or maliciously. This enforcement leads to better security and maintainability.
- **Boosts performance:** The module system allows the JVM to optimize the loading and execution of code, leading to improved start-up times, lower memory consumption, and faster execution.

The adoption of the module system has greatly improved the Java platform's maintainability, security, and performance.

Modules Example

Let's explore the module system by considering two example modules: `com.house.brickhouse` and `com.house.bricks`. The `com.house.brickhouse` module contains two classes, `House1` and `House2`, which calculate the number of bricks needed for houses with different levels. The `com.house.bricks` module contains a `Story` class that provides a method to count bricks based on the number of levels. Here's the directory structure for `com.house.brickhouse`:

```
src
── com.house.brickhouse
        ├── com
        │   └── house
        │       └── brickhouse
        │           ├── House1.java
        │           └── House2.java
        └── module-info.java
```

com.house.brickhouse:
module-info.java:

```
module com.house.brickhouse {
    requires com.house.bricks;
    exports com.house.brickhouse;
}
```

com/house/brickhouse/House1.java:

```
package com.house.brickhouse;
import com.house.bricks.Story;

public class House1 {
    public static void main(String[] args) {
        System.out.println("My single-level house will need " + Story.count(1) + " bricks");
    }
}
```

com/house/brickhouse/House2.java:

```
package com.house.brickhouse;
import com.house.bricks.Story;

public class House2 {
    public static void main(String[] args) {
        System.out.println("My two-level house will need " + Story.count(2) + " bricks");
    }
}
```

Now let's look at the directory structure for `com.house.bricks`:

```
src
└── com.house.bricks
    ├── com
    │   └── house
    │       └── bricks
    │           └── Story.java
    └── module-info.java

com.house.bricks:
   module-info.java:

module com.house.bricks {
    exports com.house.bricks;
}

com/house/bricks/Story.java:

package com.house.bricks;

public class Story {
    public static int count(int level) {
        return level * 18000;
    }
}
```

Compilation and Run Details

We compile the `com.house.bricks` module first:

```
$ javac -d mods/com.house.bricks src/com.house.bricks/module-info.java src/com.house.bricks/com/
house/bricks/Story.java
```

Next, we compile the `com.house.brickhouse` module:

```
$ javac --module-path mods -d mods/com.house.brickhouse
src/com.house.brickhouse/module-info.java
src/com.house.brickhouse/com/house/brickhouse/House1.java
src/com.house.brickhouse/com/house/brickhouse/House2.java
```

Now we run the `House1` example:

```
$ java --module-path mods -m com.house.brickhouse/com.house.brickhouse.House1
```

Output:

```
My single-level house will need 18000 bricks
```

Then we run the `House2` example:

```
$ java --module-path mods -m com.house.brickhouse/com.house.brickhouse.House2
```

Output:

```
My two-level house will need 36000 bricks
```

Introducing a New Module

Now, let's expand our project by introducing a new module that provides various types of bricks. We'll call this module `com.house.bricktypes`, and it will include different classes for different types of bricks. Here's the new directory structure for the `com.house.bricktypes` module:

```
src
└── com.house.bricktypes
    ├── com
    │   └── house
    │       └── bricktypes
    │           ├── ClayBrick.java
    │           └── ConcreteBrick.java
    └── module-info.java

com.house.bricktypes:
    module-info.java:

module com.house.bricktypes {
    exports com.house.bricktypes;
}
```

The `ClayBrick.java` and `ConcreteBrick.java` classes will define the properties and methods for their respective brick types.

`ClayBrick.java`:

```
package com.house.bricktypes;

public class ClayBrick {
    public static int getBricksPerSquareMeter() {
        return 60;
    }
}
```

`ConcreteBrick.java`:

```
package com.house.bricktypes;

public class ConcreteBrick {
    public static int getBricksPerSquareMeter() {
        return 50;
    }
}
```

With the new module in place, we need to update our existing modules to make use of these new brick types. Let's start by updating the `module-info.java` file in the `com.house.brickhouse` module:

```
module com.house.brickhouse {
    requires com.house.bricks;
    requires com.house.bricktypes;
    exports com.house.brickhouse;
}
```

We modify the `House1.java` and `House2.java` files to use the new brick types.

`House1.java`:

```
package com.house.brickhouse;
import com.house.bricks.Story;
import com.house.bricktypes.ClayBrick;

public class House1 {
    public static void main(String[] args) {
        int bricksPerSquareMeter = ClayBrick.getBricksPerSquareMeter();
        System.out.println("My single-level house will need "
                           + Story.count(1, bricksPerSquareMeter) + " clay bricks");
    }
}
```

`House2.java`:

```
package com.house.brickhouse;
import com.house.bricks.Story;
import com.house.bricktypes.ConcreteBrick;

public class House2 {
```

```
    public static void main(String[] args) {
        int bricksPerSquareMeter = ConcreteBrick.getBricksPerSquareMeter();
        System.out.println("My two-level house will need "
                           + Story.count(2, bricksPerSquareMeter) + " concrete bricks");
    }
}
```

By making these changes, we're allowing our `House1` and `House2` classes to use different types of bricks, which adds more flexibility to our program. Let's now update the `Story.java` class in the `com.house.bricks` module to accept the bricks per square meter:

```
package com.house.bricks;

public class Story {
    public static int count(int level, int bricksPerSquareMeter) {
        return level * bricksPerSquareMeter * 300;
    }
}
```

Now that we've updated our modules, let's compile and run them to see the changes in action:

- Create a new `mods` directory for the `com.house.bricktypes` module:

```
$ mkdir mods/com.house.bricktypes
```

- Compile the `com.house.bricktypes` module:

```
$ javac -d mods/com.house.bricktypes
src/com.house.bricktypes/module-info.java
src/com.house.bricktypes/com/house/bricktypes/*.java
```

- Recompile the `com.house.bricks` and `com.house.brickhouse` modules:

```
$ javac --module-path mods -d mods/com.house.bricks
src/com.house.bricks/module-info.java src/com.house.bricks/com/house/bricks/Story.java

$ javac --module-path mods -d mods/com.house.brickhouse
src/com.house.brickhouse/module-info.java
src/com.house.brickhouse/com/house/brickhouse/House1.java
src/com.house.brickhouse/com/house/brickhouse/House2.java
```

With these updates, our program is now more versatile and can handle different types of bricks. This is just one example of how the modular system in Java can make our code more flexible and maintainable.

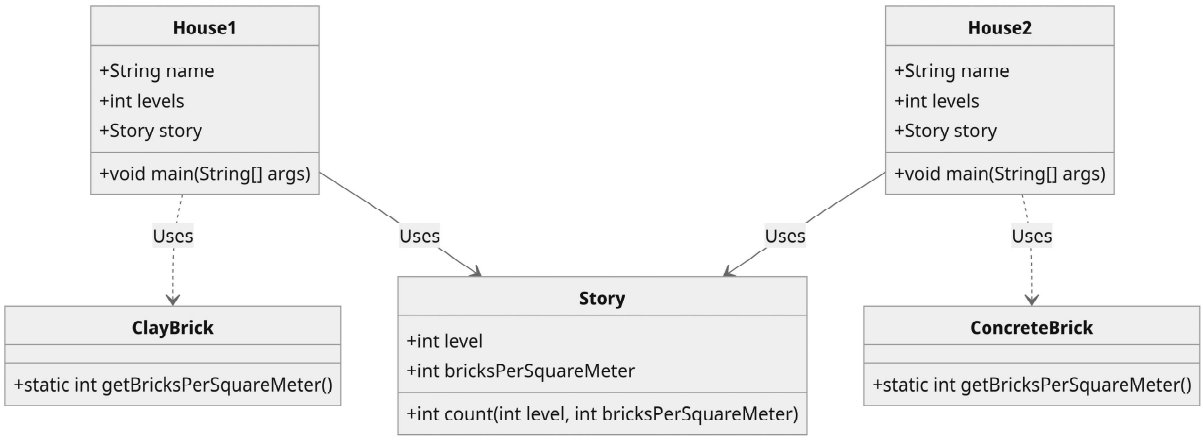

Figure 3.1 Class Diagram to Show the Relationships Between Modules

Let's now visualize these relationships with a class diagram. Figure 3.1 includes the new module `com.house.bricktypes`, and the arrows represent "Uses" relationships. `House1` uses `Story` and `ClayBrick`, whereas `House2` uses `Story` and `ConcreteBrick`. As a result, instances of `House1` and `House2` will contain references to instances of `Story` and either `ClayBrick` or `ConcreteBrick`, respectively. They use these references to interact with the methods and attributes of the `Story`, `ClayBrick`, and `ConcreteBrick` classes. Here are more details:

- `House1` and `House2`: These classes represent two different types of houses. Both classes have the following attributes:
  - `name`: A string representing the name of the house.
  - `levels`: An integer representing the number of levels in the house.
  - `story`: An instance of the `Story` class representing a level of the house.
  - `main(String[] args)`: The entry method for the class, which acts as the initial kick-starter for the application's execution.
- `Story`: This class represents a level in a house. It has the following attributes:
  - `level`: An integer representing the level number.
  - `bricksPerSquareMeter`: An integer representing the number of bricks per square meter for the level.
  - `count(int level, int bricksPerSquareMeter)`: A method that calculates the total number of bricks required for a given level and bricks per square meter.
- `ClayBrick` and `ConcreteBrick`: These classes represent two different types of bricks. Both classes have the following attributes:
  - `getBricksPerSquareMeter()`: A static method that returns the number of bricks per square meter. This method is called by the houses to obtain the value needed for calculations in the `Story` class.

Next, let's look at the use-case diagram of the Brick House Construction system with the House Owner as the actor and Clay Brick and Concrete Brick as the systems (Figure 3.2). This diagram illustrates how the House Owner interacts with the system to calculate the number of bricks required for different types of houses and choose the type of bricks for the construction.

Here's more information on the elements of the use-case diagram:

- **House Owner:** This is the actor who wants to build a house. The House Owner interacts with the Brick House Construction system in the following ways:
  - **Calculate Bricks for House 1:** The House Owner uses the system to calculate the number of bricks required to build House 1.
  - **Calculate Bricks for House 2:** The House Owner uses the system to calculate the number of bricks required to build House 2.
  - **Choose Brick Type:** The House Owner uses the system to select the type of bricks to be used for the construction.
- **Brick House Construction:** This system helps the House Owner in the construction process. It provides the following use cases:
  - **Calculate Bricks for House 1:** This use case calculates the number of bricks required for House 1. It interacts with both the Clay Brick and Concrete Brick systems to get the necessary data.
  - **Calculate Bricks for House 2:** This use case calculates the number of bricks required for House 2. It also interacts with both the Clay Brick and Concrete Brick systems to get the necessary data.
  - **Choose Brick Type:** This use case allows the House Owner to choose the type of bricks for the construction.
- **Clay Brick and Concrete Brick:** These systems provide the data (e.g., size, cost) to the Brick House Construction system that is needed to calculate the number of bricks required for the construction of the houses.

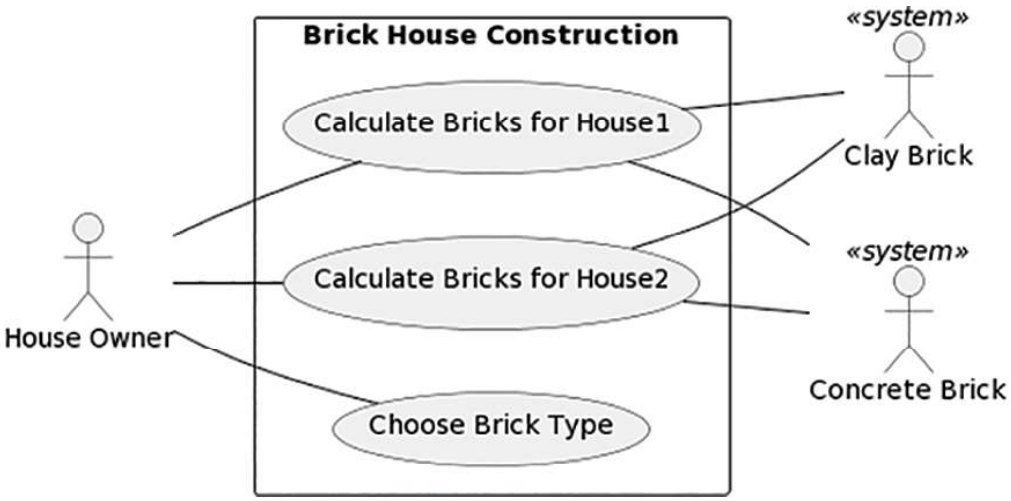

Figure 3.2 Use-Case Diagram of Brick House Construction

From Monolithic to Modular: The Evolution of the JDK

Before the introduction of the modular JDK, the bloating of the JDK led to overly complex and difficult-to-read applications. In particular, complex dependencies and cross-dependencies made it difficult to maintain and extend applications. JAR (Java Archives) hell (i.e., problems related to loading classes in Java) arose due to both the lack of simplicity and JARs' lack of awareness about the classes they contained.

The sheer footprint of the JDK also posed a challenge, particularly for smaller devices or other situations where the entire monolithic JDK wasn't needed. The modular JDK came to the rescue, transforming the JDK landscape.

Continuing the Evolution: Modular JDK in JDK 11 and Beyond

The Java Platform Module System (JPMS) was first introduced in JDK 9, and its evolution has continued in subsequent releases. JDK 11, the first long-term support (LTS) release after JDK 8, further refined the modular Java platform. Some of the notable improvements and changes made in JDK 11 are summarized here:

- **Removal of deprecated modules:** Some Java Enterprise Edition (EE) and Common Object Request Broker Architecture (CORBA) modules that had been deprecated in JDK 9 were finally removed in JDK 11. This change promoted a leaner Java platform and reduced the maintenance burden.
- **Matured module system:** The JPMS has matured over time, benefiting from the feedback of developers and real-world usage. Newer JDK releases have addressed issues, improved performance, and optimized the module system's capabilities.
- **Refined APIs:** APIs and features have been refined in subsequent releases, providing a more consistent and coherent experience for developers using the module system.
- **Continued enhancements:** JDK 11 and subsequent releases have continued to enhance the module system—for example, by offering better diagnostic messages and error reporting, improved JVM performance, and other incremental improvements that benefit developers.

Implementing Modular Services with JDK 17

With the JDK's modular approach, we can enhance the concept of services (introduced in Java 1.6) by decoupling modules that provide the service interface from their provider module, eventually creating a fully decoupled consumer. To employ services, the type is usually declared as an interface or an abstract class, and the service providers need to be clearly identified in their modules, enabling them to be recognized as providers. Lastly, consumer modules are required to utilize those providers.

To better explain the decoupling that occurs, we'll use a step-by-step example to build a `BricksProvider` along with its providers and consumers.

Service Provider

A service provider is a module that implements a service interface and makes it available for other modules to consume. It is responsible for implementing the functionalities defined in the service interface. In our example, we'll create a module called `com.example.bricksprovider`, which will implement the `BrickHouse` interface and provide the service.

Creating the `com.example.bricksprovider` Module

First, we create a new directory called `bricksprovider`; inside it, we create the `com/example/bricksprovider` directory structure. Next, we create a `module-info.java` file in the `bricksprovider` directory with the following content:

```
module com.example.bricksprovider {
    requires com.example.brickhouse;
    provides com.example.brickhouse.BrickHouse with com.example.bricksprovider.BricksProvider;
}
```

This `module-info.java` file declares that our module requires the `com.example.brickhouse` module and provides an implementation of the `BrickHouse` interface through the `com.example.bricksprovider.BricksProvider` class.

Now, we create the `BricksProvider.java` file inside the `com/example/bricksprovider` directory with the following content:

```
package com.example.bricksprovider;
import com.example.brickhouse.BrickHouse;

public class BricksProvider implements BrickHouse {
    @Override
    public void build() {
        System.out.println("Building a house with bricks...");
    }
}
```

Service Consumer

A service consumer is a module that uses a service provided by another module. It declares the service it requires in its `module-info.java` file using the `uses` keyword. The service consumer can then use the `ServiceLoader` API to discover and instantiate implementations of the required service.

Creating the `com.example.builder` Module

First, we create a new directory called `builder`; inside it, we create the `com/example/builder` directory structure. Next, we create a `module-info.java` file in the `builder` directory with the following content:

```
module com.example.builder {
    requires com.example.brickhouse;
    uses com.example.brickhouse.BrickHouse;
}
```

This `module-info.java` file declares that our module requires the `com.example.brickhouse` module and uses the `BrickHouse` service.

Now, we create a `Builder.java` file inside the `com/example/builder` directory with the following content:

```
package com.example.builder;
import com.example.brickhouse.BrickHouse;
import java.util.ServiceLoader;

public class Builder {
    public static void main(String[] args) {
        ServiceLoader<BrickHouse> loader = ServiceLoader.load(BrickHouse.class);
        loader.forEach(BrickHouse::build);
    }
}
```

A Working Example

Let's consider a simple example of a modular Java application that uses services:

- `com.example.brickhouse`: A module that defines the `BrickHouse` service interface that other modules can implement
- `com.example.bricksprovider`: A module that provides an implementation of the `BrickHouse` service and declares it in its `module-info.java` file using the `provides` keyword
- `com.example.builder`: A module that consumes the `BrickHouse` service and declares the required service in its `module-info.java` file using the `uses` keyword

The builder can then use the `ServiceLoader` API to discover and instantiate the `BrickHouse` implementation provided by the `com.example.bricksprovider` module.