Pattern Matching for `switch` and Record Patterns

Changes to the Java® Language Specification • Version 21-internal-adhoc.gbierman.20230509

This document describes changes to the Java Language Specification ⇗ to support *Pattern Matching for* `switch` and *Record Patterns*, which are both proposed features of Java SE 21. See JEP 441 ⇗ and JEP 440 ⇗ respectively for overviews of the features.

Changes are described with respect to existing sections of the JLS. New text is indicated like this and deleted text is indicated ~~like this~~. Explanation and discussion, as needed, is set aside in grey boxes.

Changelog:

2023-05-09:

- *Replaced non-denotable any patterns and the process of resolving certain type patterns to any patterns with an inferred property of type patterns.*
- *Added explicit capture of match types in record pattern type inference.*

2023-04-06: First draft. Changes from last preview version:

- *Added requirement that guards cannot assign to any variable that is not declared by the guard.*
- *Added improved support for qualified names of enum constants as case constants*
- *Removal of support for record patterns to appear in the header of an enhanced* `for` *statement.*
- *Removal of parenthesized patterns.*
- *Clarified that* `switch` *does not support selector expressions of type* `boolean`*,* `long`*,* `float`*, or* `double`*.*
- *Strengthened definition of an exhaustive* `switch` *block.*
- *Refactored grammar for switch labels to make guards part of the label production.*
- *Minor editorial improvements (including renaming downcast compatible as checked cast compatible).*

Chapter 3: Lexical Structure

3.9 Keywords

51 character sequences, formed from ASCII characters, are reserved for use as keywords and cannot be used as identifiers (3.8 ⇗). Another ~~16~~ 17 character sequences, also formed from ASCII characters, may be interpreted as keywords or as other tokens, depending on the context in which they appear.

Keyword:
ReservedKeyword
ContextualKeyword

ReservedKeyword:

(one of)

```
abstract  continue  for         new       switch
assert    default   if          package   synchronized
boolean   do        goto        private   this
break     double    implements  protected throw
byte      else      import      public    throws
case      enum      instanceof  return    transient
catch     extends   int         short     try
char      final     interface   static    void
class     finally   long        strictfp  volatile
const     float     native      super     while
```

_ (underscore)

ContextualKeyword:

(one of)

~~exports opens requires uses~~
~~module permits sealed var~~
~~non-sealed provides to with~~
~~open record transitive yield~~
exports permits sealed var
module provides to when
non-sealed record transitive with
open requires uses yield
opens

> *The `when` contextual keyword has been added.*

The keywords `const` and `goto` are reserved, even though they are not currently used. This may allow a Java compiler to produce better error messages if these C++ keywords incorrectly appear in programs.

The keyword `strictfp` is obsolete and should not be used in new code.

The keyword _ (underscore) is reserved for possible future use in parameter declarations.

`true` and `false` are not keywords, but rather boolean literals (3.10.3 ⇗).

`null` is not a keyword, but rather the null literal (3.10.8 ⇗).

During the reduction of input characters to input elements (3.5 ⇗), a sequence of input characters that notionally matches a contextual keyword is reduced to a contextual keyword if and only if both of the following conditions hold:

1. The sequence is recognized as a terminal specified in a suitable context of the syntactic grammar (2.3 ⇗), as follows:
    - For `module` and `open`, when recognized as a terminal in a *ModuleDeclaration* (7.7 ⇗).
    - For `exports`, `opens`, `provides`, `requires`, `to`, `uses`, and `with`, when recognized as a terminal in a *ModuleDirective*.
    - For `transitive`, when recognized as a terminal in a *RequiresModifier*.

        For example, recognizing the sequence `requires transitive ;` does not make use of RequiresModifier, so the term `transitive` is reduced here to an identifier and not a

contextual keyword.

- For `var`, when recognized as a terminal in a *LocalVariableType* (14.4) or a *LambdaParameterType* (15.27.1).

  In other contexts, attempting to use `var` as an identifier will cause an error, because `var` is not a TypeIdentifier (3.8).

- For `yield`, when recognized as a terminal in a *YieldStatement* (14.21).

  In other contexts, attempting to use the `yield` as an identifier will cause an error, because `yield` is neither a TypeIdentifier nor a UnqualifiedMethodIdentifier.

- For `record`, when recognized as a terminal in a *RecordDeclaration* (8.10).
- For `non-sealed`, `permits`, and `sealed`, when recognized as a terminal in a *NormalClassDeclaration* (8.1) or a *NormalInterfaceDeclaration* (9.1).
- For `when`, when recognized as a terminal in a *Guard* (14.11.1).

2. The sequence is not immediately preceded or immediately followed by an input character that matches *JavaLetterOrDigit*.

In general, accidentally omitting white space in source code will cause a sequence of input characters to be tokenized as an identifier, due to the "longest possible translation" rule (3.2). For example, the sequence of twelve input characters `p u b l i c s t a t i c` is always tokenized as the identifier `publicstatic`, rather than as the reserved keywords `public` and `static`. If two tokens are intended, they must be separated by white space or a comment.

The rule above works in tandem with the "longest possible translation" rule to produce an intuitive result in contexts where contextual keywords may appear. For example, the sequence of eleven input characters `v a r f i l e n a m e` is usually tokenized as the identifier `varfilename`, but in a local variable declaration, the first three input characters are tentatively recognized as the contextual keyword `var` by the first condition of the rule above. However, it would be confusing to overlook the lack of white space in the sequence by recognizing the next eight input characters as the identifier `filename`. (This would mean that the sequence undergoes different tokenization in different contexts: an identifier in most contexts, but a contextual keyword and an identifier in local variable declarations.) Accordingly, the second condition prevents recognition of the contextual keyword `var` on the grounds that the immediately following input character `f` is a JavaLetterOrDigit. The sequence `v a r f i l e n a m e` is therefore tokenized as the identifier `varfilename` in a local variable declaration.

As another example of the careful recognition of contextual keywords, consider the sequence of 15 input characters `n o n - s e a l e d c l a s s`. This sequence is usually translated to three tokens - the identifier `non`, the operator `-`, and the identifier `sealedclass` - but in a normal class declaration, where the first condition holds, the first ten input characters are tentatively recognized as the contextual keyword `non-sealed`. To avoid translating the sequence to two keyword tokens (`non-sealed` and `class`) rather than three non-keyword tokens, and to avoid rewarding the programmer for omitting white space before `class`, the second condition prevents recognition of the contextual keyword. The sequence `n o n - s e a l e d c l a s s` is therefore tokenized as three tokens in a class declaration.

In the rule above, the first condition depends on details of the syntactic grammar, but a compiler for the Java programming language can implement the rule without fully parsing the input program. For example, a heuristic could be used to track the contextual state of the tokenizer, as long as the heuristic guarantees that valid uses of contextual keywords are tokenized as keywords, and valid uses of identifiers are tokenized as identifiers. Alternatively, a compiler could always tokenize a contextual keyword as an identifier, leaving it to a later phase to recognize special uses of these identifiers.

Chapter 5: Conversions and Contexts

5.5 Casting Contexts

Casting contexts allow the operand of a cast expression (15.16 ⇗) to be converted to the type explicitly named by the cast operator. Compared to assignment contexts and invocation contexts, casting contexts allow the use of more of the conversions defined in 5.1 ⇗, and allow more combinations of those conversions.

If the expression is of a primitive type, then a casting context allows the use of one of the following:

- an identity conversion (5.1.1 ⇗)
- a widening primitive conversion (5.1.2 ⇗)
- a narrowing primitive conversion (5.1.3 ⇗)
- a widening and narrowing primitive conversion (5.1.4 ⇗)
- a boxing conversion (5.1.7 ⇗)
- a boxing conversion followed by a widening reference conversion (5.1.5 ⇗)

If the expression is of a reference type, then a casting context allows the use of one of the following:

- an identity conversion (5.1.1 ⇗)
- a widening reference conversion (5.1.5 ⇗)
- a widening reference conversion followed by an unboxing conversion
- a widening reference conversion followed by an unboxing conversion, then followed by a widening primitive conversion
- a narrowing reference conversion (5.1.6 ⇗)
- a narrowing reference conversion followed by an unboxing conversion
- an unboxing conversion (5.1.8 ⇗)
- an unboxing conversion followed by a widening primitive conversion

If the expression has the null type, then the expression may be cast to any reference type.

If a casting context makes use of a narrowing reference conversion that is checked or partially unchecked (5.1.6.2 ⇗, 5.1.6.3 ⇗), then a run time check will be performed on the class of the expression's value, possibly causing a `ClassCastException`. Otherwise, no run time check is performed.

If an expression can be converted to a reference type by a casting conversion *other than a narrowing reference conversion which is unchecked*, we say the expression (or its value) is ~~*downcast*~~ *checked cast* *compatible with* the reference type.

> *It is proposed to rename the relation downcast compatible to checked cast compatible. The previous terminology was confusing as it contains casts commonly referred to as upcasts.*

If an expression of reference type *S* is checked cast compatible with another reference type *T*, we say that the type *S* is *checked cast convertible* to type *T*.

The following tables enumerate which conversions are used in certain casting contexts. Each

conversion is signified by a symbol:

- - signifies no conversion allowed
- ≈ signifies identity conversion (5.1.1 ⇗)
- *ω* signifies widening primitive conversion (5.1.2 ⇗)
- *η* signifies narrowing primitive conversion (5.1.3 ⇗)
- *ωη* signifies widening and narrowing primitive conversion (5.1.4 ⇗)
- ⇑ signifies widening reference conversion (5.1.5 ⇗)
- ⇓ signifies narrowing reference conversion (5.1.6 ⇗)
- ⊕ signifies boxing conversion (5.1.7 ⇗)
- ⊗ signifies unboxing conversion (5.1.8 ⇗)

In the tables, a comma between symbols indicates that a casting context uses one conversion followed by another. The type `Object` means any reference type other than the eight wrapper classes `Boolean`, `Byte`, `Short`, `Character`, `Integer`, `Long`, `Float`, `Double`.

Table 5.5-A. Casting to primitive types

To → From ↓	**byte**	**short**	**char**	**int**	**long**	**float**	**double**	**boolean**
byte	≈	*ω*	*ωη*	*ω*	*ω*	*ω*	*ω*	-
short	*η*	≈	*η*	*ω*	*ω*	*ω*	*ω*	-
char	*η*	*η*	≈	*ω*	*ω*	*ω*	*ω*	-
int	*η*	*η*	*η*	≈	*ω*	*ω*	*ω*	-
long	*η*	*η*	*η*	*η*	≈	*ω*	*ω*	-
float	*η*	*η*	*η*	*η*	*η*	≈	*ω*	-
double	*η*	*η*	*η*	*η*	*η*	*η*	≈	-
boolean	-	-	-	-	-	-	-	≈
Byte	⊗	⊗,*ω*	-	⊗,*ω*	⊗,*ω*	⊗,*ω*	⊗,*ω*	-
Short	-	⊗	-	⊗,*ω*	⊗,*ω*	⊗,*ω*	⊗,*ω*	-
Character	-	-	⊗	⊗,*ω*	⊗,*ω*	⊗,*ω*	⊗,*ω*	-
Integer	-	-	-	⊗	⊗,*ω*	⊗,*ω*	⊗,*ω*	-
Long	-	-	-	-	⊗	⊗,*ω*	⊗,*ω*	-
Float	-	-	-	-	-	⊗	⊗,*ω*	-
Double	-	-	-	-	-	-	⊗	-
Boolean	-	-	-	-	-	-	-	⊗
Object	⇓,⊗	⇓,⊗	⇓,⊗	⇓,⊗	⇓,⊗	⇓,⊗	⇓,⊗	⇓,⊗

Table 5.5-B. Casting to reference types

To → From ↓	**Byte**	**Short**	**Character**	**Integer**	**Long**	**Float**	**Double**	**Boolean**	**Object**
byte	⊕	-	-	-	-	-	-	-	⊕,⇑
short	-	⊕	-	-	-	-	-	-	⊕,⇑
char	-	-	⊕	-	-	-	-	-	⊕,⇑
int	-	-	-	⊕	-	-	-	-	⊕,⇑
long	-	-	-	-	⊕	-	-	-	⊕,⇑
float	-	-	-	-	-	⊕	-	-	⊕,⇑
double	-	-	-	-	-	-	⊕	-	⊕,⇑
boolean	-	-	-	-	-	-	-	⊕	⊕,⇑
Byte	≈	-	-	-	-	-	-	-	⇑
Short	-	≈	-	-	-	-	-	-	⇑
Character	-	-	≈	-	-	-	-	-	⇑
Integer	-	-	-	≈	-	-	-	-	⇑
Long	-	-	-	-	≈	-	-	-	⇑
Float	-	-	-	-	-	≈	-	-	⇑
Double	-	-	-	-	-	-	≈	-	⇑
Boolean	-	-	-	-	-	-	-	≈	⇑
Object	⇓	⇓	⇓	⇓	⇓	⇓	⇓	⇓	≈

Example 5.5-1. Casting for Reference Types

```
class Point { int x, y; }
interface Colorable { void setColor(int color); }
class ColoredPoint extends Point implements Colorable {
    int color;
    public void setColor(int color) { this.color = color; }
}
final class EndPoint extends Point {}

class Test {
    public static void main(String[] args) {
        Point p = new Point();
        ColoredPoint cp = new ColoredPoint();
        Colorable c;
        // The following may cause errors at run time because
        // we cannot be sure they will succeed; this possibility
        // is suggested by the casts:
        cp = (ColoredPoint)p;  // p might not reference an
                               // object which is a ColoredPoint
                               // or a subclass of ColoredPoint
        c = (Colorable)p;      // p might not be Colorable
        // The following are incorrect at compile time because
        // they can never succeed as explained in the text:
        Long l = (Long)p;            // compile-time error #1
```

```
        EndPoint e = new EndPoint();
        c = (Colorable)e;            // compile-time error #2
    }
}
```

Here, the first compile-time error occurs because the class types `Long` and `Point` are unrelated (that is, they are not the same, and neither is a subclass of the other), so a cast between them will always fail.

The second compile-time error occurs because a variable of type `EndPoint` can never reference a value that implements the interface `Colorable`. This is because `EndPoint` is a `final` type, and a variable of a `final` type always holds a value of the same run-time type as its compile-time type. Therefore, the run-time type of variable `e` must be exactly the type `EndPoint`, and type `EndPoint` does not implement `Colorable`.

Example 5.5-2. Casting for Array Types

```
class Point {
    int x, y;
    Point(int x, int y) { this.x = x; this.y = y; }
    public String toString() { return "("+x+","+y+")"; }
}
interface Colorable { void setColor(int color); }
class ColoredPoint extends Point implements Colorable {
    int color;
    ColoredPoint(int x, int y, int color) {
        super(x, y); setColor(color);
    }
    public void setColor(int color) { this.color = color; }
    public String toString() {
        return super.toString() + "@" + color;
    }
}

class Test {
    public static void main(String[] args) {
        Point[] pa = new ColoredPoint[4];
        pa[0] = new ColoredPoint(2, 2, 12);
        pa[1] = new ColoredPoint(4, 5, 24);
        ColoredPoint[] cpa = (ColoredPoint[])pa;
        System.out.print("cpa: {");
        for (int i = 0; i < cpa.length; i++)
            System.out.print((i == 0 ? " " : ", ") + cpa[i]);
        System.out.println(" }");
    }
}
```

This program compiles without errors and produces the output:

```
cpa: { (2,2)@12, (4,5)@24, null, null }
```

Example 5.5-3. Casting Incompatible Types at Run Time

```
class Point { int x, y; }
interface Colorable { void setColor(int color); }
class ColoredPoint extends Point implements Colorable {
    int color;
    public void setColor(int color) { this.color = color; }
}
```

```
class Test {
    public static void main(String[] args) {
        Point[] pa = new Point[100];

        // The following line will throw a ClassCastException:
        ColoredPoint[] cpa = (ColoredPoint[])pa;
        System.out.println(cpa[0]);
        int[] shortvec = new int[2];
        Object o = shortvec;

        // The following line will throw a ClassCastException:
        Colorable c = (Colorable)o;
        c.setColor(0);
    }
}
```

This program uses casts to compile, but it throws exceptions at run time, because the types are incompatible.

Chapter 6: Names

6.3 Scope of a Declaration

6.3.1 Scope for Pattern Variables in Expressions

6.3.1.6 `switch` Expressions

The following ~~rule applies~~ rules apply to a `switch` expression ~~(15.28)~~ with a switch block consisting of switch rules (14.11.1):

- A pattern variable introduced by a switch label is definitely matched in the associated switch rule expression, switch rule block, or switch rule `throw` statement.

  It is a compile-time error if any pattern variable introduced by a switch label is already in scope at the associated switch rule expression, switch rule block, or switch rule `throw` statement.

The following rules apply to a switch expression with a switch block consisting of switch labeled statement groups (14.11.1):

- A pattern variable introduced by a switch label is definitely matched in all the statements of the switch labeled statement group.

  It is a compile-time error if any pattern variable introduced by a switch label is already in scope at the statements of the switch labeled statement group.

- A pattern variable introduced by a statement *S* contained in a switch labeled statement group ~~(14.11.1)~~ is definitely matched at all the statements following *S*, if any, in the switch labeled statement group.

6.3.2 Scope for Pattern Variables in Statements

6.3.2.6 `switch` Statements

The following ~~rule applies~~ rules apply to a `switch` statement ~~(14.11)~~ with a switch block consisting of switch rules (14.11.1):

- A pattern variable introduced by a switch label is definitely matched in the associated switch rule expression, switch rule block, or switch rule `throw` statement.

  It is a compile-time error if any pattern variable introduced by a switch label is already in scope at the associated switch rule expression, switch rule block, or switch rule `throw` statement.

The following rules apply to a switch expression with a switch block consisting of switch labeled statement groups (14.11.1):

- A pattern variable introduced by a switch label is definitely matched in all the statements of the switch labeled statement group.

  It is a compile-time error if any pattern variable introduced by a switch label is already in scope at the statements of the switch labeled statement group.

- A pattern variable introduced by a labeled statement *S* contained in a switch block statement group ~~(14.11.1)~~ is definitely matched at all the statements following *S*, if any, in the switch block statement group.

6.3.3 Scope for Pattern Variables in Patterns

6.3.3.1 Record Patterns

The following rule applies to a record pattern *p*:

- For each pattern *q* in the nested pattern list of *p*, a pattern variable declared by *q* is definitely matched in every record pattern component that comes after it within the list.

  It is a compile-time error if a pattern variable declared by *q* is already in scope in a record pattern component that comes after it within the list.

  > *This rule enforces a linearity constraint that pattern variables can only be declared at most once in a single nested pattern list. Specifying that two record components should have equal values can not be encoded directly in the pattern but should be handled in subsequent code:*
  >
  > ```
  > Object o = ...
  > if (o instanceof Point(int x, int y) && (x == y)) { // Not the pattern Point(int x, int x)!
  >     System.out.println("Point on the diagonal");
  > }
  > ```

6.3.4 Scope for Pattern Variables in Switch Labels

Pattern variables can be introduced by `case` labels with a `case` pattern, either by the pattern itself or by a guard, and are in scope for the relevant parts of the associated `switch` expression (6.3.1.6) or `switch` statement (6.3.2.6).

The following rules apply to `case` labels:

- A pattern variable is introduced by a `case` label with a `case` pattern *p* if it is declared by *p*.
- A pattern variable declared by the pattern of a guarded `case` pattern is definitely matched in the associated guard.

  It is a compile-time error if any pattern variable declared by the pattern of a guarded `case` pattern is already in scope at its guard.
- A pattern variable is introduced by a guarded `case` label if it is introduced by the associated

guard when true (6.3.1 ⇗).

Chapter 13: Binary Compatibility

13.4 Evolution of Classes

13.4.2 `sealed`, `non-sealed`, and `final` Classes

13.4.2.1 `sealed` Classes

If a class that was freely extensible (8.1.1.2 ⇗) is changed to be declared `sealed`, then an `IncompatibleClassChangeError` is thrown if a binary of a pre-existing subclass of this class is loaded and is not a permitted direct subclass of this class (8.1.6 ⇗); such a change is not recommended for widely distributed classes.

Changing a class that was declared `final` to be declared `sealed` does not break compatibility with pre-existing binaries.

Adding a class to the set of permitted direct subclasses of a `sealed` class will not break compatibility with pre-existing binaries.

Note that evolving a `sealed` class by adding a permitted direct subclass is considered a binary compatible change because pre-existing binaries that previously linked without error (e.g., a class file that contains an exhaustive `switch` (14.11.1)) will continue to link without error. A class file that contains an exhaustive `switch` will not fail to link if the `sealed` class that it switches over is expanded by the hierarchy's owner to have a new permitted direct subclass. The JVM is not required to perform exhaustiveness checks when linking a class file that contains an exhaustive `switch`.

The execution of an exhaustive `switch` can fail with an error (a `MatchException` is thrown) if it encounters an instance of a permitted direct subclass that was not known at compile time (14.11.3, 15.28.2). Strictly speaking, the error is not flagging a binary incompatible change of the `sealed` class, but more accurately a migration incompatible change of the `sealed` class.

If a class is removed from the set of permitted direct subclasses of a `sealed` class, then an `IncompatibleClassChangeError` is thrown if the pre-existing binary of the removed class is loaded.

Deleting the `sealed` modifier from a class that does not have a `sealed` direct superclass or a `sealed` direct superinterface does not break compatibility with pre-existing binaries.

If a sealed class C did have a `sealed` direct superclass or a `sealed` direct superinterface, then deleting the `sealed` modifier would prevent C from being recompiled, as every class with a `sealed` direct superclass or a `sealed` direct superinterface must be either `final`, `sealed`, or `non-sealed`.

13.4.26 Evolution of Enum Classes

Adding or reordering enum constants in an enum class will not break compatibility with pre-existing binaries.

As with `sealed` classes (13.4.2.1), although adding an enum constant to an enum class is considered a binary compatible change, it may cause the execution of an exhaustive `switch` (14.11.1) to fail if the `switch` encounters the new enum constant that was not known at compile time (14.11.3, 15.28.2).

Deleting an enum constant from an enum class will delete the `public` field that corresponds to the enum constant (8.9.3 ⇗). The consequences are specified in 13.4.8 ⇗. Such a change is not recommended for widely distributed enum classes.

In all other respects, the binary compatibility rules for enum classes are identical to those for normal classes.

13.5 Evolution of Interfaces

13.5.2 `sealed` and `non-sealed` Interfaces

If an interface that was freely extensible (9.1.1.4 ⇗) is changed to be declared `sealed`, then an `IncompatibleClassChangeError` is thrown if a binary of a pre-existing subclass or subinterface of this interface is loaded and is not a permitted direct subclass or subinterface of this interface (9.1.4 ⇗); such a change is not recommended for widely distributed classes.

Adding a class or interface to the set of permitted direct subclasses or subinterfaces, respectively, of a `sealed` interface will not break compatibility with pre-existing binaries.

As with `sealed` classes (13.4.2.1), whilst adding a permitted direct subclass or subinterface of a `sealed` interface is considered a binary compatible change, it may cause the execution of an exhaustive `switch` (14.11.1) to fail with an error (a `MatchException` may be thrown) if the `switch` encounters an instance of the new permitted direct subclass or subinterface that was not known at compile time (14.11.3, 15.28.2).

If a class or interface is removed from the set of permitted direct subclasses or subinterfaces of a `sealed` interface, then an `IncompatibleClassChangeError` is thrown if the pre-existing binary of the removed class or interface is loaded.

Changing an interface that was declared `sealed` to be declared `non-sealed` does not break compatibility with pre-existing binaries.

> *A `non-sealed` interface I must have a `sealed` direct superinterface. Deleting the `non-sealed` modifier would prevent I from being recompiled, as every interface with a `sealed` direct superinterface must be `sealed` or `non-sealed`.*

Deleting the `sealed` modifier from an interface that does not have a `sealed` direct superinterface does not break compatibility with pre-existing binaries.

> *If a sealed interface I did have a `sealed` direct superinterface, then deleting the `sealed` modifier would prevent I from being recompiled, as every interface with a `sealed` direct superinterface must be `sealed` or `non-sealed`.*

Chapter 14: Blocks, Statements, and Patterns

14.11 The `switch` Statement

The `switch` statement transfers control to one of several statements or expressions, depending on the value of an expression.

> *SwitchStatement:*
> `switch (` *Expression* `)` *SwitchBlock*

The *Expression* is called the *selector expression*. The type of the selector expression must be `char`, `byte`, `short`, `int`, ~~`Character`, `Byte`, `Short`, `Integer`, `String`, or an enum type (8.9 ⇗),~~ or a reference type, or a compile-time error occurs.

14.11.1 Switch Blocks

The body of both a `switch` statement and a `switch` expression (15.28) is called a *switch block*.

This subsection presents general rules which apply to all switch blocks, whether they appear in `switch` statements or `switch` expressions. Other subsections present additional rules which apply either to switch blocks in `switch` statements (14.11.2) or to switch blocks in `switch` expressions (15.28.1).

SwitchBlock:
`{` *SwitchRule* {*SwitchRule*} `}`
`{` {*SwitchBlockStatementGroup*} {*SwitchLabel* `:`} `}`

SwitchRule:
SwitchLabel `->` *Expression* `;`
SwitchLabel `->` *Block*
SwitchLabel `->` *ThrowStatement*

SwitchBlockStatementGroup:
SwitchLabel `:` { *SwitchLabel* `:`} *BlockStatements*

SwitchLabel:
`case` *CaseConstant* {`,` *CaseConstant*}
`case null` [`,` `default`]
`case` *CasePattern* [*Guard*]
`default`

CaseConstant:
ConditionalExpression

CasePattern:
Pattern

Guard:
`when` *Expression*

A switch block can consist of either:

- *Switch rules*, which use `->` to introduce either a *switch rule expression*, a *switch rule block*, or a *switch rule* `throw` *statement*; or
- *Switch labeled statement groups*, which use `:` to introduce *switch labeled block statements*.

Every switch rule and switch labeled statement group starts with a *switch label*, which is either a `case` label or a `default` label. Multiple switch labels are permitted for a switch labeled statement group.

~~A `case` label has one or more `case` constants. Every `case` constant must be either a constant expression (15.29) or the name of an enum constant (8.9.1), or a compile-time error occurs.~~

~~Switch labels and their `case` constants are said to be *associated* with the switch block. No two of the `case` constants associated with a switch block may have the same value, or a compile-time error occurs.~~

A `case` label consists of either a list of `case` constants or a single `case` pattern.

Every `case` constant must be either (1) the `null` literal, (2) a constant expression (15.29), or (3) the (simple or qualified) name of an enum constant (8.9.1); otherwise a compile-time error

occurs. A single `null` case constant may also be paired with the `default` keyword.

A `case` label with a `case` pattern may have an optional `when` expression, known as a *guard*, which represents a further test on values that match the pattern. A `case` label is said to be *unguarded* if either (i) it has no guard, or (ii) it has a guard that is a constant expression (15.29) with value `true`; and *guarded* otherwise.

Switch labels and their `case` constants and `case` patterns are said to be *associated* with the switch block.

For a given switch block both of the following must be true, otherwise a compile-time error occurs:

- No two of the `case` constants associated with a switch block may have the same value.
- No more than one `default` label may be associated with a switch block.

Any guard associated with a `case` label must have type `boolean` or `Boolean`. Any variable that is used but not declared by a guard must either be `final` or effectively final (4.12.4) and cannot be assigned to (15.26), incremented (15.14.2), or decremented (15.14.3), otherwise a compile-time error occurs. It is a compile-time error if a guard is a constant expression (15.29) with the value `false`.

~~The switch block of a `switch` statement or a `switch` expression is *compatible* with the type of the selector expression, *T*, if both of the following are true:~~

- ~~If *T* is not an enum type, then every `case` constant associated with the switch block is assignment compatible with *T* (5.2).~~
- ~~If *T* is an enum type, then every `case` constant associated with the switch block is an enum constant of type *T*.~~

The switch block of a `switch` statement or a `switch` expression is *switch compatible* with the type of the selector expression, *T*, if all of the following are true:

- If any `null` constant is associated with the switch block, then *T* is a reference type.
- If *T* is an enum type, then every `case` constant associated with the switch block that is the name of an enum constant is the name of an enum constant of type *T*.
- If *T* is not an enum type, then every `case` constant associated with the switch block that is the name of of an enum constant is the qualified name of an enum constant that is assignment compatible with *T* (5.2).
- Every `case` constant associated with the switch block that is a constant expression is assignment compatible with *T*, and *T* is one of `char`, `byte`, `short`, `int`, `Character`, `Byte`, `Short`, `Integer`, or `String`.
- Every pattern *p* associated with the switch block, *p* is applicable for type *T* (14.30.3).

Switch blocks are not designed to work with the types `boolean`, `long`, `float`, and `double`. The selector expression of a `switch` statement or `switch` expression can not have one of these types.

The switch block of a `switch` statement or a `switch` expression must be switch compatible with the type of the selector expression, or a compile-time error occurs.

A switch label in a switch block is said to be *dominated* if for every value that it applies to, one of

the preceding switch labels would also apply. It is a compile-time error if any switch label in a switch block is dominated. The rules for determining whether a switch label is dominated are as follows:

- A `case` label with a `case` pattern *q* is dominated if there is a preceding unguarded `case` label in the switch block with a `case` pattern *p*, and *p* dominates *q* (14.30.3).

  The definition of one pattern dominating another pattern is based on types. For example, the type pattern Object o dominates the type pattern String s, and so the following results in a compile-time error:

  ```
  Object obj = ...
  switch (obj) {
      case Object o ->
          System.out.println("An object");
      case String s  ->                   // Error - dominated case label
          System.out.println("A string");
  }
  ```

  A guarded `case` label with a `case` pattern is dominated by a `case` label with the same pattern but without the guard. For example, the following results in a compile-time error:

  ```
  String str = ...;
  switch (str) {
      case String s ->
          System.out.println("A string");
      case String s when s.length() == 2 ->  // Error - dominated case label
          System.out.println("Two character string");
      ...
  }
  ```

  On the other hand, a guarded `case` label with a `case` pattern is not considered to dominate an unguarded `case` label with the same `case` pattern. This allows the following common pattern programming style:

  ```
  Integer j = ...;
  switch (j) {
      case Integer i when i <= 0 ->
          System.out.println("Less than or equal to zero");
      case Integer i ->
          System.out.println("An integer");
  }
  ```

  The only exception is where the guard is a constant expression that has the value `true`, for example:

  ```
  Integer j = ...;
  switch (j) {
      case Integer i when true ->             // Allowed but why write this?
          System.out.println("An integer");
      case Integer i ->                       // Error - dominated case label
          System.out.println("An integer");
  }
  ```

- A `case` label with a `case` constant *c* is dominated if one of the following holds:
  - *c* is a constant expression of a primitive type *S*, and there is a preceding `case` label in the switch block with a `case` pattern *p*, where *p* is unconditional for the wrapper class

of *S*.

- *c* is either a constant expression of a reference type *T*, and there is a preceding `case` label in the switch block with a `case` pattern *p*, where *p* is unconditional for the type *T*.
- *c* is an enum constant of type *T*, and there is a preceding `case` label in the switch block with a `case` pattern *p*, where *p* is unconditional for the type *T*.

For example, a `case` label with an `Integer` type pattern dominates a `case` label with an integer literal:

```
Integer j = ...;
switch (j) {
    case Integer i ->
        System.out.println("An integer");
    case 42 ->                              // Error - dominated!
        System.out.println("42!");
}
```

Analysis of guards—undecidable in general—is not attempted. For example, the following results in a compile-time error, even though the first switch label does not match if the value of the selector expression is `42`:

```
Integer j = ...;
switch (j) {
    case Integer i when i != 42 ->
        System.out.println("An integer that isn't 42");
    case 42 ->                                  // Error - dominated!
        System.out.println("42!");
}
```

Any `case` labels with `case` constants should appear before those with `case` patterns; for example:

```
Integer j = ...;
switch (j) {
    case 42 ->
        System.out.println("42");
    case Integer i when i < 50 ->
        System.out.println("An integer less than 50");
    case Integer i  ->
        System.out.println("An integer");
}
```

- A `case` label with a `case` pattern is dominated if there is a preceding `default` label in the switch block.
- A `case` label with a `null` case constant is dominated if there is a preceding `default` label in the switch block.

  If used, a `default` label should come last in a `switch` block.

  For historical reasons, a `default` label may appear before `case` labels that do not have a `null` `case` constant or a `case` pattern.

```
int i = ...;
switch(i) {
    default ->
```

```
        System.out.println("Some other integer");
    case 42 -> // allowed
        System.out.println("42");
}
```

This style is discouraged in new code.

- A switch label is dominated if there is a preceding `case null, default` label in the switch block.

  If used, a `case null, default` label always comes last in a `switch` block.

- A `default` label is dominated if there is a preceding unguarded `case` label in the switch block with a `case` pattern *p* where *p* is unconditional for the type of the selector expression (14.30.3).
- A `case null, default` label is dominated if there is a preceding unguarded `case` label in the switch block with a `case` pattern *p* where *p* is unconditional for the type of the selector expression (14.30.3).

  A `case` label with a `case` pattern that is unconditional for the type of the selector expression will, as the name suggests, match every value and so behaves like a `default` label. A switch block can not have more than one switch label that acts like a `default`.

It is a compile-time error if, in a switch block that consists of switch labeled statement groups, a statement is labeled with a `case` pattern that declares one or more pattern variables, and either:

- An immediately preceding statement in the switch block can complete normally (14.22 ⇗), or
- The statement is labeled with more than one switch label.

The first condition prevents a statement group from "falling through" to another statement group without initializing pattern variables. For example, were a statement labeled by `case Integer i` reachable from the preceding statement group, the pattern variable `i` would not have been initialized:

```
Object o = "Hello";
switch (o) {
    case String s:
        System.out.println("String: " + s );  // No break!
    case Integer i:
        System.out.println(i + 1);             // Error! Can be reached
                                               // without matching the
                                               // pattern `Integer i`
    default:
}
```

Switch blocks consisting of switch label statement groups allow multiple labels to apply to a statement group. The second condition prevents a statement group from being executed based on one label without initializing the pattern variables of another label. For example:

```
Object o = "Hello World";
switch (o) {
    case String s:
    case Integer i:
        System.out.println(i + 1);  // Error! Can be reached
                                    // without matching the
                                    // pattern `Integer i`
```

```
    default:
}

Object obj = null;
switch (obj) {
    case null:
    case String s:
        System.out.println(s);       // Error! Can be reached
                                     // without matching the
                                     // pattern `String s`
    default:
}
```

Both of these conditions apply only when the `case` pattern declares pattern variables. The following examples, in contrast, are unproblematic:

```
record R() {}
record S() {}

Object o = "Hello World";
switch (o) {
    case String s:
        System.out.println(s);       // No break!
    case R():
        System.out.println("It's either an R or a string");
        break;
    default:
}

Object ob = new R();
switch (ob) {
    case R():
    case S():                        // Multiple case labels!
        System.out.println("Either R or an S");
        break;
    default:
}

Object obj = null;
switch (obj) {
    case null:
    case R():                        // Multiple case labels!
        System.out.println("Either null or an R");
        break;
    default:
}
```

14.11.1.1 Exhaustive Switch Blocks

The switch block of a `switch` expression or `switch` statement is *exhaustive* for a selector expression *e* if one of the following cases applies:

- There is a `default` label associated with the switch block.
- There is a `case` label with a `default` associated with the switch block.
- The set containing all the `case` constants and `case` patterns appearing in an unguarded `case` label (collectively known as `case` elements) associated with the switch block is non-

empty and covers the type of the selector expression *e*.

A set of `case` elements, *P*, *covers* a type *T* if one of the following cases applies:

- *P* covers a type *U* where *T* and *U* have the same erasure.
- *P* contains a pattern that is unconditional for *T*.
- *T* is a type variable with upper bound *B* and *P* covers *B*.
- *T* is an intersection type $T_1 \& \ldots \& T_n$ and *P* covers T_i, for one of the types T_i ($1 \le i \le n$).
- The type *T* is an enum class type *E* and *P* contains all of the enum constants of *E*.

  Note this means that a `default` label is permitted, but not required in the case where all the enum constants appear as `case` constants. For example:

  ```
  enum E { F, G, H }

  static int testEnumExhaustive(E e) {
      return switch(e) {
              case F -> 0;
              case G -> 1;
              case H -> 2;     // No default required!
          };
  }
  ```

- The type *T* names an `abstract sealed` class or `sealed` interface *C* and for every permitted direct subclass or subinterface *D* of *C*, one of the following two conditions holds:

  1. There is no type that both names *D* and is a subtype of *T*, or
  2. There is a type *U* that both names *D* and is a subtype of *T*, and *P* covers *U*.

  Note this means that a `default` label is permitted, but not required in the case where the switch block exhausts all the permitted direct subclasses and subinterfaces of an `abstract sealed` class or `sealed` interface. For example:

  ```
  sealed interface I permits A, B, C {}
  final class A  implements I {}
  final class B  implements I {}
  record C(int j) implements I {}  // Implicitly final

  static int testExhaustive1(I i) {
      return switch(i) {
              case A a -> 0;
              case B b -> 1;
              case C c -> 2;            // No default required!
          };
  }
  ```

  As the switch block contains switch labels supporting patterns that match against values of type `A`, `B` and `C`, and no other instances of type `I` are permitted, this switch block is exhaustive.

  The fact that a permitted direct subclass or subinterface may only extend a particular parameterization of a generic `sealed` superclass or superinterface means that it may not always need to be considered when determining whether a switch block is exhaustive. For example:

  ```
  sealed interface J<X> permits D, E {}
  final class D<Y> implements J<String> {}
  final class E<X> implements J<X> {}
  ```

```
static int testExhaustive2(J<Integer> ji) {
    return switch(ji) {          // Exhaustive!
        case E<Integer> e -> 42;
    };
}
```

As the selector expression has type `J<Integer>` the permitted direct subclass `D` need not be considered as there is no possibility that the value of `ji` can be an instance of `D`.

- The type *T* names a record class *R*, and *P* contains a record pattern *p* with a type that names *R* and for every record component of *R* of type *U*, if any, the singleton set containing the corresponding nested component pattern of *p* covers *U*.

  This case covers the case where a record pattern whose nested pattern list contains patterns that all cover their corresponding component type is considered to cover the record type. For example:

  ```
  record Test<X>(Object o, X x){}

  static int testExhaustiveRecordPattern(Test<String> r) {
      return switch(r) {                    // Exhaustive!
          case Test<String>(Object o, String s) -> 0;
      };
  }
  ```

- *P* rewrites to a set *Q* and *Q* covers *T*.

  A set of case elements, *P*, *rewrites* to the set *Q*, if a subset of *P* reduces to a pattern *p*, and *Q* consists of the remaining elements of *P* along with the pattern *p*.

  A non-empty set of patterns, *RP*, *reduces* to a single pattern *rp* if one of the following holds:

  - *RP* covers some type *U*, and *rp* is a type pattern of type *U*.
  - *RP* consists of record patterns whose types all erase to the same record class *R* with *k* (*k*≥*1*) components and there is a distinguished component c_r (*1*≤*r*≤*k*) of *R* such that for every other component c_i (*1*≤*i*≤*k*, *i*≠*r*) the set containing the nested patterns from the record patterns corresponding to component c_i collapses to a single pattern q_i, the set containing the nested patterns from the record patterns corresponding to the component c_r reduces to a single pattern q, and *rp* is the record pattern of type *R* with a nested pattern list consisting of the patterns $q_1, \ldots, q_{r-1}, q, q_{r+1}, \ldots, q_k$.

    A non-empty set of patterns *CP collapses* to a single pattern *cp* if one of the following holds:

    - *CP* consists of type patterns whose types all have the same erasure *T*, and *cp* is a type pattern of type *T*.
    - *CP* consists of record patterns whose types all erase to the same record class *R* with *k* (*k*≥*1*) components and for every component the set containing the corresponding nested patterns from the record patterns collapses to a single pattern q_j (*1*≤*j*≤*k*), and *cp* is the record pattern of type *R* with a nested pattern list consisting of the patterns $q_1,\ldots q_k$.

  Ordinarily record patterns match only a subset of the values of the record type. However, a number of record patterns in a switch block can combine to actually match all of the values of the record type. For example:

```
sealed interface I permits A, B, C {}
final class A   implements I {}
final class B   implements I {}
record C(int j) implements I {}  // Implicitly final
record Box(I i) {}

int testExhaustiveRecordPatterns(Box b) {
    return switch (b) {      // Exhaustive!
            case Box(A a) -> 0;
            case Box(B b) -> 1;
            case Box(C c) -> 2;
    };
}
```

Determining whether this switch block is exhaustive requires the analysis of the combination of the record patterns. The set { `Box(I i)` } covers the type `Box`, and the original set of patterns { `Box(A a)`, `Box(B b)`, `Box(C c)` } can be rewritten to the set { `Box(I i)` }. This is because the set { `A a`, `B b`, `C c` } reduces to the pattern `I i` (because the same set covers the type `I`), and thus the set { `Box(A a)`, `Box(B b)`, `Box(C c)` } reduces to the pattern `Box(I i)`.

However, rewriting a set of record patterns is not so simple. For example:

```
record IPair(I i, I j){}

int testNonExhaustiveRecordPatterns(IPair p) {
    return switch (p) {      // Not Exhaustive!
            case IPair(A a, A a) -> 0;
            case IPair(B b, B b) -> 1;
            case IPair(C c, C c) -> 2;
    };
}
```

It is tempting to apply the logic from the previous example to rewrite the set of patterns { `IPair(A a, A a)`, `IPair(B b, B b)`, `IPair(C c, C c)` } to the set { `IPair(I i, I j)` }, and hence conclude that the switch block exhausts the type `IPair`. But this is incorrect as, for example, the switch block does not actually have a label that matches an `IPair` value whose first component is an `A` value, and second component is a `B` value. It is only valid to combine record patterns on one component if they match the same values in the other components. For example, the set containing the three record patterns `IPair(A a, I i)`, `IPair(B b, I i)`, and `IPair(C c, I i)` can be reduced to the pattern `IPair(I j, I i)`.

A `switch` statement or expression is *exhaustive* if its switch block is exhaustive for the selector expression.

14.11.1.2 Determining which Switch Label Applies at Run-Time

~~Both the execution of a `switch` statement (14.11.3) and the evaluation of a `switch` expression (15.28.2) need to determine if a switch label *matches* the value of the selector expression. To determine whether a switch label in a switch block matches a given value, the value is compared with the `case` constants associated with the switch block. Then:~~

- ~~If one of the `case` constants is equal to the value, then we say that the `case` label which contains the `case` constant *matches*.~~

  ~~Equality is defined in terms of the == operator (15.21) unless the value is a `String`, in which case equality is defined in terms of the `equals` method of class `String`.~~

- ~~If no `case` label matches but there is a `default` label, then we say that the `default` label~~

~~*matches*.~~

Both the execution of a `switch` statement (14.11.3) and the evaluation of a `switch` expression (15.28.2) need to determine if a switch label associated with the switch block *applies* to the value of the selector expression. This proceeds as follows:

1. If the value is the null reference, then a `case` label with a `null case` constant applies.
2. If the value is not the null reference, then we determine the first (if any) `case` label in the switch block that applies to the value as follows:
    - A `case` label with a non-null `case` constant *c* applies to a value of type `Character`, `Byte`, `Short`, or `Integer`, if the value is first subjected to unboxing conversion (5.1.8⇗) and the constant *c* is equal to the unboxed value.

      Any unboxing conversion must complete normally as the value being unboxed is guaranteed not to be the null reference.

      Equality is defined in terms of the `==` operator (15.21⇗).
    - A `case` label with a non-null `case` constant *c* applies to a value that is not of type `Character`, `Byte`, `Short`, or `Integer`, if the constant *c* is equal to the value.

      Equality is defined in terms of the `==` operator unless the value is a `String`, in which case equality is defined in terms of the `equals` method of class `String`.
    - Determining that a `case` label with a `case` pattern *p* applies to a value proceeds first by checking if the value matches the pattern *p* (14.30.2).

      If pattern matching completes abruptly then the process of determining which switch label applies completes abruptly for the same reason.

      If pattern matching succeeds and the `case` label is unguarded then this `case` label applies.

      If pattern matching succeeds and the `case` label is guarded, then the guard is evaluated. If the result is of type `Boolean`, it is subjected to unboxing conversion (5.1.8⇗).

      If evaluation of the guard or the subsequent unboxing conversion (if any) completes abruptly for some reason, the process of determining which switch label applies completes abruptly for the same reason.

      Otherwise, if the resulting value is `true` then the `case` label applies.
    - A `case null, default` label applies to every value
3. If the value is not the null reference, and no `case` label applies according to the rules of step 2, then if a `default` label is associated with the switch block, that label applies.

A single `case` label can contain several `case` constants. The label ~~matches~~ applies to the value of the selector expression if any one of its constants ~~matches~~ is equal to the value of the selector expression. For example, in the following code, the `case` label matches if the enum variable `day` is either one of the enum constants shown:

```
switch (day) {
    ...
    case SATURDAY, SUNDAY :
        System.out.println("It's the weekend!");
```

```
            break;
        ...
}
```

If a `case` label with a `case` pattern applies, then this is because the process of pattern matching the value against the pattern has succeeded (14.30.2). If a value successfully matches a pattern then the process of pattern matching initializes any pattern variables declared by the pattern.

For historical reasons, a `default` label is only considered after all `case` labels have failed to match, even if some of those labels appear after the `default` label. However, subsequent labels may only make use of non-`null` case constants (14.11.1), and as a matter of style, programmers are encouraged to place their `default` labels last.

~~*`null` cannot be used as a `case` constant because it is not a constant expression. Even if `case` `null` was allowed, it would be undesirable because the code in that `case` can never be executed. Namely, if the selector expression is of a reference type (that is, `String` or a boxed primitive type or an enum type), then an exception will occur if the selector expression evaluates to `null` at run time. In the judgment of the designers of the Java programming language, propagating the exception is a better outcome than either having no `case` label match, or having the `default` label match.*~~

In C and C++ the body of a `switch` statement can be a statement and statements with `case` labels do not have to be immediately contained by that statement. Consider the simple loop:

```
for (i = 0; i < n; ++i) foo();
```

where `n` is known to be positive. A trick known as Duff's device can be used in C or C++ to unroll the loop, but this is not valid code in the Java programming language:

```
int q = (n+7)/8;
switch (n%8) {
    case 0: do { foo();     // Great C hack, Tom,
    case 7:      foo();     // but it's not valid here.
    case 6:      foo();
    case 5:      foo();
    case 4:      foo();
    case 3:      foo();
    case 2:      foo();
    case 1:      foo();
            } while (--q > 0);
}
```

Fortunately, this trick does not seem to be widely known or used. Moreover, it is less needed nowadays; this sort of code transformation is properly in the province of state-of-the-art optimizing compilers.

14.11.2 The Switch Block of a `switch` Statement

In addition to the general rules for switch blocks (14.11.1), there are further rules for switch blocks in `switch` statements.

An *enhanced* `switch` statement is one where either (i) the type of the selector expression is not `char`, `byte`, `short`, `int`, `Character`, `Byte`, `Short`, `Integer`, `String`, or an enum type, or (ii) there is a `case` pattern or `null case` constant associated with the switch block.

~~Namely, all~~ All of the following must be true for the switch block of a `switch` statement, or a compile-time error occurs:

- ~~No more than one `default` label is associated with the `switch` block.~~

This condition is now covered by the dominance requirements.

- Every switch rule expression in the switch block is a statement expression (14.8 ⇗).

  `switch` statements differ from `switch` expressions in terms of which expressions may appear to the right of an arrow (`->`) in the switch block, that is, which expressions may be used as switch rule expressions. In a `switch` statement, only a statement expression may be used as a switch rule expression, but in a `switch` expression, any expression may be used (15.28.1).

- If the `switch` statement is an enhanced `switch` statement, then it must be exhaustive (14.11.1.1).

Prior to Java SE 20, `switch` statements (and `switch` expressions) were limited in two ways: (i) the type of the selector expression was restricted to either an integral type (excluding `long`), an enum type, or `String`; and (ii) only non-null `case` constants were supported. Moreover, unlike `switch` expressions, `switch` statements did not have to be exhaustive. This is often the cause of difficult-to-detect bugs, where no switch label applies and the `switch` statement will silently do nothing. For example:

```
enum E { A, B, C }

E e = ...;
switch (e) {
    case A -> System.out.println("A");
    case B -> System.out.println("B");
    // No case for C!
}
```

With Java SE 20, `switch` statements have been enhanced in the sense that along with supporting `case` patterns, the two limitations listed above have also been lifted. The designers of the Java programming language decided that enhanced `switch` statements should align with `switch` expressions and be required to be exhaustive. This is often achieved with the addition of a trivial `default` label. For example, the following enhanced `switch` statement is not exhaustive:

```
Object o = ...;
switch (o) {     // Error - non-exhaustive switch!
    case String s -> System.out.println("A string!");
}
```

but it can easily be made exhaustive:

```
Object o = ...;
switch (o) {
    case String s -> System.out.println("A string!");
    default -> {}
}
```

For compatibility reasons, `switch` statements that are not enhanced `switch` statements are not required to be exhaustive.

14.11.3 Execution of a `switch` Statement

A `switch` statement is executed by first evaluating the selector expression. ~~Then:~~ If evaluation of the selector expression completes abruptly, then the entire `switch` statement completes abruptly for the same reason.

- ~~Otherwise, if the result of evaluating the selector expression is `null`, then a~~

~~`NullPointerException` is thrown and the entire `switch` statement completes abruptly for that reason.~~

- ~~Otherwise, if the result of evaluating the selector expression is of type `Character`, `Byte`, `Short`, or `Integer`, it is subjected to unboxing conversion (5.1.8). If this conversion completes abruptly, the entire `switch` statement completes abruptly for the same reason.~~

If evaluation of the selector expression completes normally ~~and the result is non-`null`, and the subsequent unboxing conversion (if any) completes normally~~, then execution of the `switch` statement continues by determining if a switch label associated with the switch block ~~matches~~ applies to the value of the selector expression (14.11.1.2). Then:

- If the process of determining which switch label applies completes abruptly, then the entire `switch` statement completes abruptly for the same reason.
- If no switch label ~~matches, the entire `switch` statement completes normally.~~ applies, then one of the following holds:
  - If the value of the selector expression is `null`, then a `NullPointerException` is thrown and the entire `switch` statement completes abruptly for that reason.
  - If the `switch` statement is an enhanced `switch` statement, then a `MatchException` is thrown and the entire `switch` statement completes abruptly for that reason.
  - If the value of the selector expression is not `null`, and the `switch` statement is not an enhanced `switch` statement, then the entire `switch` statement completes normally.
- If a switch label ~~matches~~ applies, then one of the following ~~applies~~ holds:
  - If it is the switch label for a switch rule expression, then the switch rule expression is necessarily a statement expression (14.11.2). The statement expression is evaluated. If the evaluation completes normally, then the `switch` statement completes normally. If the result of evaluation is a value, it is discarded.
  - If it is the switch label for a switch rule block, then the block is executed. If this block completes normally, then the `switch` statement completes normally.
  - If it is the switch label for a switch rule `throw` statement, then the `throw` statement is executed.
  - If it is the switch label for a switch labeled statement group, then all the statements in the switch block that follow the switch label are executed in order. If these statements complete normally, then the `switch` statement completes normally.
  - Otherwise, there are no statements that follow the ~~matched~~ switch label that applies in the switch block, and the `switch` statement completes normally.

If execution of any statement or expression in the switch block completes abruptly, it is handled as follows:

- If execution of a statement completes abruptly because of a `break` with no label, then no further action is taken and the `switch` statement completes normally.

  Abrupt completion because of a `break` with a label is handled by the general rule for labeled statements (14.7).

- If execution of a statement or expression completes abruptly for any other reason, then the

`switch` statement completes abruptly for the same reason.

Abrupt completion because of a `yield` statement is handled by the general rule for switch expressions (15.28.2).

Example 14.11.3-1. Fall-Through in the `switch` Statement

When a ~~selector expression matches a switch label~~ switch label applies, and that switch label is for a switch rule, the switch rule expression or statement introduced by the switch label is executed, and nothing else. In the case of a switch label for a statement group, all the block statements in the switch block that follow the switch label are executed, including those that appear after subsequent switch labels. The effect is that, as in C and C++, execution of statements can "fall through labels."

For example, the program:

```
class TooMany {
    static void howMany(int k) {
        switch (k) {
            case 1: System.out.print("one ");
            case 2: System.out.print("too ");
            case 3: System.out.println("many");
        }
    }
    public static void main(String[] args) {
        howMany(3);
        howMany(2);
        howMany(1);
    }
}
```

contains a `switch` block in which the code for each `case` falls through into the code for the next `case`. As a result, the program prints:

```
many
too many
one too many
```

Fall through can be the cause of subtle bugs. If code is not to fall through `case` to `case` in this manner, then `break` statements can be used to indicate when control should be transferred, or switch rules can be used, as in the program:

```
class TwoMany {
    static void howMany(int k) {
        switch (k) {
            case 1: System.out.println("one");
                    break;  // exit the switch
            case 2: System.out.println("two");
                    break;  // exit the switch
            case 3: System.out.println("many");
                    break;  // not needed, but good style
        }
    }
    static void howManyAgain(int k) {
        switch (k) {
            case 1 -> System.out.println("one");
            case 2 -> System.out.println("two");
            case 3 -> System.out.println("many");
        }
    }
```

```
        public static void main(String[] args) {
            howMany(1);
            howMany(2);
            howMany(3);
            howManyAgain(1);
            howManyAgain(2);
            howManyAgain(3);
        }
    }
```

This program prints:

```
one
two
many
one
two
many
```

14.30 Patterns

A *pattern* describes a test that can be performed on a value. Patterns appear as operands of statements and expressions, which provide the values to be tested. Patterns declare zero or more local variables, known as *pattern variables*.

The process of testing a value against a pattern is known as *pattern matching*. If a value successfully matches a pattern, then the process of pattern matching initializes the pattern ~~variable~~ variables, if any, declared by the pattern.

Pattern variables are only in scope (6.3 ⇗) where pattern matching succeeds and thus the pattern variables will have been initialized. It is not possible to use a pattern variable that has not been initialized.

14.30.1 Kinds of Patterns

A *type pattern* is used to test whether a value is an instance of the type appearing in the pattern. A *record pattern* is used to test whether a value is an instance of a record class type and, if it is, to recursively perform pattern matching on the record component values.

Pattern:
    TypePattern
    RecordPattern

TypePattern:
    LocalVariableDeclaration

RecordPattern:
    ReferenceType ([*PatternList*])

PatternList :
    Pattern { , *Pattern* }

The following productions from 4.3 ⇗, 8.3 ⇗, 8.4.1 ⇗, and 14.4 ⇗ are shown here for convenience:

LocalVariableDeclaration:
    {VariableModifier} LocalVariableType VariableDeclaratorList

VariableModifier:
    Annotation

```
final
```

LocalVariableType:
UnannType
`var`

VariableDeclaratorList:
VariableDeclarator {, VariableDeclarator}

VariableDeclarator:
VariableDeclaratorId [= VariableInitializer]

VariableDeclaratorId:
Identifier [Dims]

Dims:
{Annotation} `[ ]` *{{Annotation}* `[ ]`*}*

See 8.3 for UnannType.

A pattern that does not appear as an element in the nested pattern list of a record pattern is called a *top-level* pattern; otherwise it is called a *nested* pattern.

A type pattern declares exactly one pattern variable. The *Identifier* in the local variable declaration specifies the name of the pattern variable.

The rules for a pattern variable declared in a type pattern are specified in 14.4. In addition, all of the following must be true, or a compile-time error occurs:

- The *LocalVariableType* in a top-level type pattern denotes a reference type (and furthermore is not `var`).
- The *VariableDeclaratorList* consists of a single *VariableDeclarator*.
- The *VariableDeclarator* has no initializer.
- The *VariableDeclaratorId* has no bracket pairs.

The type of a pattern variable declared in a top-level type pattern is the reference type denoted by *LocalVariableType*.

The type of a pattern variable declared in a nested type pattern is determined as follows:

- If the *LocalVariableType* is *UnannType* then the type of the pattern variable is denoted by *UnannType*
- If the *LocalVariableType* is `var` then the type pattern must be nested in a pattern list of a record pattern with type *R*. Let *T* be the type of the corresponding component field in *R*. The type of the pattern variable is the upward projection of *T* with respect to all synthetic type variables mentioned by *T*.

  Consider the following record declaration:

  ```
  record R<T>(ArrayList<T> r){}
  ```

  Given the pattern `R<String>(var r)`*, the type of the pattern variable* `r` *is thus* `ArrayList<String>`*.*

~~The type of a type pattern is the type of its pattern variable.~~

A type pattern is said to be *null-matching* if it appears as an element in a pattern list of a record pattern with type *R*, the corresponding record component of *R* has type *U*, and the type pattern is

unconditional for the type *U* (14.30.3).

Note that this compile-time property of type patterns is used in the run-time process of pattern matching (14.30.2), so it is associated with the type pattern for use at run time.

A record pattern consists of a *ReferenceType* and a nested pattern list. If *ReferenceType* is not a record class type (8.10 ⇗) then a compile-time error occurs.

If the *ReferenceType* is a raw type, then the type of the record pattern is inferred, as described in 18.5.5. It is a compile-time error if no type can be inferred for the record pattern.

Otherwise, the type of the record pattern is *ReferenceType*.

The length of the record pattern's nested pattern list must be the same as the length of the record component list in the declaration of the record class named by *ReferenceType*; otherwise a compile-time error occurs.

Currently, there is no support for variable arity record patterns. This may be supported in future versions of the Java Programming Language.

A record pattern declares the pattern variables, if any, that are declared by the patterns in the nested pattern list.

The following relation has been moved to 14.30.3.

~~An expression *e* is *compatible* with a pattern of type *T* if *e* is downcast compatible with *T* (5.5).~~

~~*Compatibility of an expression with a pattern is used by the `instanceof` pattern match operator (15.20.2).*~~

14.30.2 Pattern Matching

Pattern matching is the process of testing a value against a pattern at run time. Pattern matching is distinct from statement execution (14.1 ⇗) and expression evaluation (15.1 ⇗). If a value successfully matches a pattern, then the process of pattern matching will initialize all the pattern variables declared by the pattern, if any.

The process of pattern matching may involve expression evaluation or statement execution. Accordingly, pattern matching is said to *complete abruptly* if evaluation of a expression or execution of a statement completes abruptly. An abrupt completion always has an associated reason, which is always a `throw` with a given value. Pattern matching is said to *complete normally* if it does not complete abruptly.

The rules for determining whether a value matches a pattern, and for initializing pattern variables, are as follows:

- The null reference *matches* a type pattern if the type pattern is null-matching (14.30.1); and *does not match* otherwise.

  If the null reference matches, then the pattern variable declared by the type pattern is initialized to the null reference.

  If the null reference does not match, then the pattern variable declared by the type pattern is not initialized.

- A value *v* that is not the null reference *matches* a type pattern of type *T* if *v* can be cast to *T* without raising a `ClassCastException`; and *does not match* otherwise.

  If *v* matches, then the pattern variable declared by the type pattern is initialized to *v*.

  If *v* does not match, then the pattern variable declared by the type pattern is not initialized.

- The null reference *does not match* a record pattern.

  In this case, any pattern variables declared by the record pattern are not initialized.

- A value *v* that is not the null reference *matches* a record pattern with type *R* and nested pattern list *L* if (i) *v* can be cast to *R* without raising a `ClassCastException`; and (ii) each record component of *v* matches the corresponding pattern in *L*; and *does not match* otherwise.

  Each record component of *v* is determined by invoking the accessor method of *v* corresponding to that component. If execution of the invocation of the accessor method completes abruptly for reason *S*, then pattern matching completes abruptly by throwing a `MatchException` with cause *S*.

  > *Any pattern variable declared in a pattern appearing in the nested pattern list is initialized only if all the patterns in the list match.*

~~*There is no rule to cover a value that is the null reference. This is because the solitary construct that performs pattern matching, the `instanceof` pattern match operator (15.20.2), only does so when a value is not the null reference. It is possible that future versions of the Java programming language will allow pattern matching in other expressions and statements.*~~

14.30.3 Properties of Patterns

A pattern *p* is said to be *applicable* at a type *T* if one of the following rules apply:

- A type pattern that declares a pattern variable of a reference type *U* is applicable at another reference type *T* if *T* is checked cast convertible to *U* (5.5).
- A type pattern that declares a pattern variable of a primitive type *P* is applicable at the type *P*.
- A record pattern with type *R* and nested pattern list *L* is applicable at type *T* if (i) *T* is checked cast convertible to *R*, and (ii) for every pattern *p* appearing in *L*, if any, *p* is applicable at the type of the corresponding component field in *R* (5.5).

A pattern *p* is said to be *unconditional* for a type *T* if every value of type *T* will match *p*, and so the pattern matching could be elided. It is defined as follows:

- A type pattern that declares a pattern variable of a reference type *S* is unconditional for another reference type *T* if the erasure of *T* is a subtype of the erasure of *S*.
- A type pattern that declares a pattern variable of a primitive type *P* is unconditional for the type *P*.

> *Note that no record pattern is unconditional because the null reference does not match any record pattern.*

A pattern *p* is said to *dominate* another pattern *q* if every value that matches *q* also matches *p*, and is defined as follows:

- A pattern *p* dominates a type pattern that declares a pattern variable of type *T* if *p* is unconditional for *T*.
- A pattern *p* dominates a record pattern with type *R* if *p* is unconditional for *R*.
- A record pattern with type *R* and nested pattern list *L* dominates another record pattern with type *S* and nested pattern list *M* if (i) the erasure of *S* is a subtype of the erasure of *R*, and (ii) every pattern, if any, in *L* dominates the corresponding pattern in *M*.

Chapter 15: Expressions

15.6 Normal and Abrupt Completion of Evaluation

Every expression has a normal mode of evaluation in which certain computational steps are carried out. The following sections describe the normal mode of evaluation for each kind of expression.

If all the steps are carried out without an exception being thrown, the expression is said to *complete normally*.

If, however, evaluation of an expression throws an exception, then the expression is said to *complete abruptly*. An abrupt completion always has an associated reason, which is always a `throw` with a given value.

Run-time exceptions are thrown by the predefined operators as follows:

- A class instance creation expression (15.9.4), array creation expression (15.10.2), method reference expression (15.13.3), array initializer expression (10.6), string concatenation operator expression (15.18.1), or lambda expression (15.27.4) throws an `OutOfMemoryError` if there is insufficient memory available.
- An array creation expression (15.10.2) throws a `NegativeArraySizeException` if the value of any dimension expression is less than zero.
- An array access expression (15.10.4) throws a `NullPointerException` if the value of the array reference expression is `null`.
- An array access expression (15.10.4) throws an `ArrayIndexOutOfBoundsException` if the value of the array index expression is negative or greater than or equal to the `length` of the array.
- A field access expression (15.11) throws a `NullPointerException` if the value of the object reference expression is `null`.
- A method invocation expression (15.12) that invokes an instance method throws a `NullPointerException` if the target reference is `null`.
- A cast expression (15.16) throws a `ClassCastException` if a cast is found to be impermissible at run time.
- An integer division (15.17.2) or integer remainder (15.17.3) operator throws an `ArithmeticException` if the value of the right-hand operand expression is zero.
- An assignment to an array component of reference type (15.26.1), a method invocation expression (15.12), or a prefix or postfix increment (15.14.2, 15.15.1) or decrement operator (15.14.3, 15.15.2) may all throw an `OutOfMemoryError` as a result of boxing conversion (5.1.7).
- An assignment to an array component of reference type (15.26.1) throws an `ArrayStoreException` when the value to be assigned is not compatible with the component type of the array (10.5).
- A switch expression (15.28) throws an ~~`IncompatibleClassChangeError`~~ `MatchException` if no switch label matches the value of the selector expression.

A method invocation expression can also result in an exception being thrown if an exception

occurs that causes execution of the method body to complete abruptly.

A class instance creation expression can also result in an exception being thrown if an exception occurs that causes execution of the constructor to complete abruptly.

Various linkage and virtual machine errors may also occur during the evaluation of an expression. By their nature, such errors are difficult to predict and difficult to handle.

If an exception occurs, then evaluation of one or more expressions may be terminated before all steps of their normal mode of evaluation are complete; such expressions are said to complete abruptly.

If evaluation of an expression requires evaluation of a subexpression, then abrupt completion of the subexpression always causes the immediate abrupt completion of the expression itself, with the same reason, and all succeeding steps in the normal mode of evaluation are not performed.

The terms "complete normally" and "complete abruptly" are also applied to the execution of statements (14.1 ⇗). A statement may complete abruptly for a variety of reasons, not just because an exception is thrown.

15.20 Relational Operators

15.20.2 The `instanceof` Operator

An `instanceof` expression may perform either type comparison or pattern matching.

InstanceofExpression:
RelationalExpression `instanceof` *ReferenceType*
RelationalExpression `instanceof` *Pattern*

If the operand to the right of the `instanceof` keyword is a *ReferenceType*, then the `instanceof` keyword is the *type comparison operator*.

If the operand to the right of the `instanceof` keyword is a *Pattern*, then the `instanceof` keyword is the *pattern match operator*.

The following rules apply when `instanceof` is the type comparison operator:

- The type of the expression *RelationalExpression* must be a reference type or the null type, or a compile-time error occurs.
- The *RelationalExpression* must be ~~downcast~~ checked cast compatible with the *ReferenceType* (5.5), or a compile-time error occurs.
- At run time, the result of the type comparison operator is determined as follows:
  - If the value of the *RelationalExpression* is the null reference (4.1 ⇗), then the result is `false`.
  - If the value of the *RelationalExpression* is not the null reference, then the result is `true` if the value could be cast to the *ReferenceType* without raising a `ClassCastException`, and `false` otherwise.

The following rules apply when `instanceof` is the pattern match operator:

- The type of the expression *RelationalExpression* must be a reference type or the null type, or a compile-time error occurs.
- ~~The *RelationalExpression* must be compatible with the *Pattern* (14.30.1), or a compile-time~~

~~error occurs.~~ The *Pattern* must be applicable at the type of the expression *RelationalExpression* (14.30.3), or a compile-time error occurs.

- ~~If the type of the *RelationalExpression* is a subtype of the type of the *Pattern*, then a compile-time error occurs.~~
- At run time, the result of the pattern match operator is determined as follows:
  - If the value of the *RelationalExpression* is the null reference, then the result is `false`.
  - If the value of the *RelationalExpression* is not the null reference, then the result is `true` if the value matches the *Pattern* (14.30.2), and `false` otherwise.

    A side effect of a `true` result is that all the pattern variables declared in Pattern, if any, will be initialized.

Example 15.20.2-1. The Type Comparison Operator

```
class Point   { int x, y; }
class Element { int atomicNumber; }
class Test {
    public static void main(String[] args) {
        Point   p = new Point();
        Element e = new Element();
        if (e instanceof Point) {  // compile-time error
            System.out.println("I get your point!");
            p = (Point)e;  // compile-time error
        }
    }
}
```

This program results in two compile-time errors. The cast `(Point)e` is incorrect because no instance of `Element` or any of its possible subclasses (none are shown here) could possibly be an instance of any subclass of `Point`. The `instanceof` expression is incorrect for exactly the same reason. If, on the other hand, the class `Point` were a subclass of `Element` (an admittedly strange notion in this example):

```
class Point extends Element { int x, y; }
```

then the cast would be possible, though it would require a run-time check, and the `instanceof` expression would then be sensible and valid. The cast `(Point)e` would never raise an exception because it would not be executed if the value of `e` could not correctly be cast to type `Point`.

Prior to Java SE 16, the ReferenceType operand of a type comparison operator was required to be reifiable (4.7 ⇗). This prevented the use of a parameterized type unless all its type arguments were wildcards. The requirement was lifted in Java SE 16 to allow more parameterized types to be used. For example, in the following program, it is legal to test whether the method parameter `x`, with static type `List<Integer>`, has a more "refined" parameterized type `ArrayList<Integer>` at run time:

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

class Test2 {
    public static void main(String[] args) {
        List<Integer> x = new ArrayList<Integer>();

        if (x instanceof ArrayList<Integer>) {  // OK
            System.out.println("ArrayList of Integers");
        }
        if (x instanceof ArrayList<String>) {  // error
```

```
                System.out.println("ArrayList of Strings");
            }
            if (x instanceof ArrayList<Object>) {  // error
                System.out.println("ArrayList of Objects");
            }
        }
    }
```

The first `instanceof` expression is legal because there is a casting conversion from `List<Integer>` to `ArrayList<Integer>`. However, the second and third `instanceof` expressions both cause a compile-time error because there is no casting conversion from `List<Integer>` to `ArrayList<String>` or `ArrayList<Object>`.

15.28 `switch` Expressions

A `switch` expression transfers control to one of several statements or expressions, depending on the value of an expression; all possible values of that expression must be handled, and all of the several statements and expressions must produce a value for the result of the `switch` expression.

SwitchExpression:
`switch` `(` *Expression* `)` *SwitchBlock*

The *Expression* is called the *selector expression*. The type of the selector expression must be `char`, `byte`, `short`, `int`, ~~`Character`, `Byte`, `Short`, `Integer`, `String`, or an enum type (8.9 ⇗),~~ or a reference type, or a compile-time error occurs.

The body of both a `switch` expression and a `switch` statement (14.11) is called a switch block. General rules which apply to all switch blocks, whether they appear in `switch` expressions or `switch` statements, are given in 14.11.1. The following productions from 14.11.1 are shown here for convenience:

SwitchBlock:
`{` SwitchRule {SwitchRule} `}`
`{` {SwitchBlockStatementGroup} {SwitchLabel `:`} `}`

SwitchRule:
SwitchLabel `->` Expression `;`
SwitchLabel `->` Block
SwitchLabel `->` ThrowStatement

SwitchBlockStatementGroup:
SwitchLabel `:` { SwitchLabel `:`} BlockStatements

~~*SwitchLabel:*~~
~~*`case` CaseConstant {`,` CaseConstant}*~~
~~*`default`*~~

SwitchLabel:
`case` CaseConstant {`,` CaseConstant }
`case null` [`,` `default`]
`case` CasePattern [Guard]
`default`

CaseConstant:
ConditionalExpression

CasePattern:
Pattern

Guard:
`when` Expression

15.28.1 The Switch Block of a `switch` Expression

In addition to the general rules for switch blocks (14.11.1), there are further rules for switch blocks in `switch` expressions.

~~Namely, all of the following must be true for the switch block of a `switch` expression, or a compile-time error occurs:~~

- ~~If the type of the selector expression is not an enum type, then there is exactly one `default` label associated with the switch block.~~
- ~~If the type of the selector expression is an enum type, then (i) the set of the `case` constants associated with the switch block includes every enum constant of the enum type, and (ii) at most one `default` label is associated with the switch block.~~

  ~~*A `default` label is permitted, but not required, when the `case` labels cover all the enum constants.*~~

~~If~~ It is a compile-time error if the switch block of a `switch` expression consists of switch rules, ~~then any switch rule block cannot~~ but one or more switch rule blocks can complete normally (14.22 ⇗).

~~If~~ It is a compile-time error if the switch block of a `switch` expression consists of switch labeled statement groups, ~~then~~ but the last statement in the switch block ~~cannot~~ can complete normally, ~~and~~ or the switch block ~~does not have any~~ has one or more switch labels after the last switch labeled statement group.

It is a compile-time error if a `switch` expression is not exhaustive (14.11.1.1).

~~*`switch` expressions cannot have empty switch blocks, unlike `switch` statements.*~~

This is covered by the result expression requirement, below.

~~*Furthermore,*~~ *`switch` expressions differ from `switch` statements in terms of which expressions may appear to the right of an arrow (`->`) in the switch block, that is, which expressions may be used as switch rule expressions. In a `switch` expression, any expression may be used as a switch rule expression, but in a `switch` statement, only a statement expression may be used (14.11.1).*

The *result expressions* of a `switch` expression are determined as follows:

- If the switch block consists of switch rules, then each switch rule is considered in turn:
  - If the switch rule is of the form `... ->` *Expression* then *Expression* is a result expression of the `switch` expression.
  - If the switch rule is of the form `... ->` *Block* then every expression which is immediately contained in a `yield` statement in *Block* whose yield target is the given `switch` expression, is a result expression of the `switch` expression.
- If the switch block consists of switch labeled statement groups, then every expression immediately contained in a `yield` statement in the switch block whose yield target is the given `switch` expression, is a result expression of the `switch` expression.

It is a compile-time error if a `switch` expression has no result expressions.

A `switch` expression is a poly expression if it appears in an assignment context or an invocation context (5.2, 5.3). Otherwise, it is a standalone expression.

Where a poly `switch` expression appears in a context of a particular kind with target type *T*, its result expressions similarly appear in a context of the same kind with target type *T*.

A poly `switch` expression is compatible with a target type *T* if each of its result expressions is compatible with *T*.

The type of a poly `switch` expression is the same as its target type.

The type of a standalone `switch` expression is determined as follows:

- If the result expressions all have the same type (which may be the null type (4.1)), then that is the type of the `switch` expression.
- Otherwise, if the type of each result expression is `boolean` or `Boolean`, then an unboxing conversion (5.1.8) is applied to each result expression of type `Boolean`, and the `switch` expression has type `boolean`.
- Otherwise, if the type of each result expression is convertible to a numeric type (5.1.8), then the type of the `switch` expression is the result of general numeric promotion (5.6) applied to the result expressions.
- Otherwise, boxing conversion (5.1.7) is applied to each result expression that has a primitive type, after which the type of the `switch` expression is the result of applying capture conversion (5.1.10) to the least upper bound (4.10.4) of the types of the result expressions.

15.28.2 Run-Time Evaluation of `switch` Expressions

A `switch` expression (14.11.1.2) is evaluated by first evaluating the selector expression. ~~Then:~~ If evaluation of the selector expression completes abruptly, then evaluation of the `switch` expression completes abruptly for the same reason.

- ~~Otherwise, if the result of evaluating the selector expression is `null`, then a `NullPointerException` is thrown and the entire `switch` expression completes abruptly for that reason.~~
- ~~Otherwise, if the result of evaluating the selector expression is of type `Character`, `Byte`, `Short`, or `Integer`, it is subjected to unboxing conversion (5.1.8). If this conversion completes abruptly, then the entire `switch` expression completes abruptly for the same reason.~~

If evaluation of the selector expression completes normally ~~and the result is non-`null`, and the subsequent unboxing conversion (if any) completes normally,~~ then evaluation of the `switch` expression continues by determining if a switch label associated with the switch block ~~matches~~ applies to the value of the selector expression (14.11.1.2). Then:

- If the process of determining which switch label applies completes abruptly, then the entire `switch` expression completes abruptly for the same reason.
- If no switch label ~~matches~~ applies, ~~and the result of evaluating the selector expression is of an enum type then an `IncompatibleClassChangeError` is thrown and the entire `switch` expression completes abruptly for that reason.~~ then one of the following holds:

    - If the value of the selector expression is `null`, then a `NullPointerException` is thrown and evaluation of the `switch` expression completes abruptly for that reason.
    - Otherwise, a `MatchException` is thrown and evaluation of the `switch` expression completes abruptly for that reason.
- If a switch label ~~matches~~ applies, then one of the following ~~applies~~ holds:
    - If it is the switch label for a switch rule expression, then the expression is evaluated. If the result of evaluation is a value, then the `switch` expression completes normally with the same value.
    - If it is the switch label for a switch rule block, then the block is executed. If this block completes normally, then the `switch` expression completes normally.
    - If it is the switch label for a switch rule `throw` statement, then the `throw` statement is executed.
    - Otherwise, all the statements in the switch block after the ~~matching~~ switch label that applies are executed in order. If these statements complete normally, then the `switch` expression completes normally.

If execution of any statement or expression in the switch block completes abruptly, it is handled as follows:

- If ~~execution~~ evaluation of an expression completes abruptly, then evaluation of the `switch` expression completes abruptly for the same reason.
- If execution of a statement completes abruptly because of a `yield` with value *V*, then the `switch` expression completes normally and the value of the `switch` expression is *V*.
- If execution of a statement completes abruptly for any reason other than a `yield` with a value, then evaluation of the `switch` expression completes abruptly for the same reason.

Chapter 16: Definite Assignment

16.2 Definite Assignment and Statements

16.2.9 `switch` Statements

- *V* is [un]assigned after a `switch` statement (14.11) iff all of the following are true:
    - *V* is [un]assigned before every `break` statement (14.15 ⇗) that may exit the `switch` statement.
    - For each switch rule (14.11.1) in the switch block, *V* is [un]assigned after the switch rule expression, switch rule block, or switch rule `throw` statement introduced by the switch rule.
    - If there is a switch labeled statement group in the switch block, then *V* is [un]assigned after the last block statement of the last switch labeled statement group.
    - If ~~there is no `default` label in the switch block~~ the switch statement is not exhaustive, or if the switch block ends with a switch label followed by the `}` separator, then *V* is [un]assigned after the selector expression.

- *V* is [un]assigned before the selector expression of a `switch` statement iff *V* is [un]assigned before the `switch` statement.
- *V* is [un]assigned before the switch rule expression, switch rule block, or switch rule `throw` statement introduced by a switch rule in the switch block iff *V* is [un]assigned after the selector expression of the `switch` statement.
- *V* is [un]assigned before the first block statement of a switch labeled statement group in the switch block iff both of the following are true:
  - *V* is [un]assigned after the selector expression of the `switch` statement.
  - If the switch labeled statement group is not the first in the switch block, *V* is [un]assigned after the last block statement of the preceding switch labeled statement group.
- *V* is [un]assigned before a block statement that is not the first of a switch labeled statement group in the switch block iff *V* is [un]assigned after the preceding block statement.

Chapter 18: Type Inference

18.5 Uses of Inference

18.5.5 Record Pattern Type Inference

When a record pattern (14.30.1) for a generic record class R appears in a context in which values of a type T will be matched against it, and the pattern does not provide type arguments for R, the type arguments are inferred, as described below.

1. If T is not checked cast convertible (5.5) to the raw type R, inference fails.
2. Otherwise, where $P_1, \ldots, P_n$ ($n \geq 1$) are the type parameters of R, let $a_1, \ldots, a_n$ be inference variables. An initial bound set, B_0, is generated from the declared bounds of $P_1, \ldots, P_n$, as described in 18.1.3.
3. A type T' is derived from T, as follows:
   - If T is a parameterized type, let T_c be the result of capture conversion (5.1.10) applied to T, and let $Z_1, \ldots, Z_k$ ($k \geq 0$) be the type variables produced by capture that are type arguments in T_c. (This includes type variables produced by the capture conversion in this step, and type variables produced by capture conversion elsewhere.) Let $\beta_1, \ldots, \beta_k$ ($k \geq 0$) be inference variables, and let θ be the substitution $[Z_1 := \beta_1, \ldots, Z_k := \beta_k]$. T' is $T_c\,\theta$.

     Additional bounds for $\beta_1, \ldots, \beta_k$ are incorporated into B_0 to form a bound set B_1, as follows:
     - If β_i ($1 \leq i \leq k$) replaced a type variable with an upper bound U, then the bound $\beta_i <: U\,\theta$ appears in the bound set
     - If β_i ($1 \leq i \leq k$) replaced a type variable with a lower bound L, then the bound $L\,\theta <: \beta_i$ appears in the bound set
     - If no proper upper bounds otherwise exist for β_i ($1 \leq i \leq k$), the bound $\beta_i <:$ `Object` appears in the bound set

- If T is any other class or interface type, then T' is the same as T, and B_1 is the same as B_0.
- If T is a type variable or an intersection type, then for each upper bound of the type variable or element of the intersection type, this step and step 4 are repeated recursively. All bounds produced in steps 3 and 4 are incorporated into a single bound set.

4. If T' is a parameterization of a generic class G, and there exists a supertype of $R<a_1, ..., a_n>$ that is also a parameterization of G, let R' be that supertype. The constraint formula $\langle T' = R' \rangle$ is reduced (18.2 ⇗) and the resulting bounds are incorporated into B_1 to produce a new bound set, B_2.

   Otherwise, B_2 is the same as B_1.

   If B_2 contains the bound *false*, inference fails.

5. Otherwise, the inference variables $a_1, ..., a_n$ are resolved in B_2 (18.4 ⇗). Unlike normal resolution, in this case resolution skips the step that attempts to produce an instantiation for an inference variable from its proper lower bounds or proper upper bounds; instead, any new instantiations are created by skipping directly to the step that introduces fresh type variables.

   If resolution fails, then inference fails.

6. Otherwise, let $A_1, ..., A_n$ be the resolved instantiations for $a_1, ..., a_n$, and let $Y_1, ..., Y_p$ ($p \geq 0$) be any fresh type variables introduced by resolution.

   The type of the record pattern is the upward projection of $R<A_1, ..., A_n>$ with respect to $Y_1, ..., Y_p$ (4.10.5 ⇗).

Example 18.5.5-1. Record Pattern Type Inference

The following program infers a parameterization for a record class:

```
record Mapper<T>(T in, T out) implements UnaryOperator<T> {
    public T apply(T arg) { return in.equals(arg) ? out : null; }
}

void test(UnaryOperator<? extends CharSequence> op) {
    if (op instanceof Mapper(var in, var out)) {
        boolean shorter = out.length() < in.length();
    }
}
```

In this case, R is the record class `Mapper`, and T is the type `UnaryOperator<? extends CharSequence>`. T is checked cast convertible to raw `Mapper`, so we'll infer an instantiation for α in `Mapper<α>`. T' is the type `UnaryOperator<β>`, where β has upper bound `CharSequence`.

`Mapper<α>` has the supertype `UnaryOperator<α>`, so we'll reduce the constraint formula `<UnaryOperator<β> = UnaryOperator<α>>`. This leads to the bound α = β. Incorporation further infers that α `<:` `CharSequence`.

Now we resolve α, yielding α = `Y`, a fresh type variable with upper bound `CharSequence`. Finally, we find the upward projection of `Mapper<Y>` with respect to `Y`, inferring that the type of the record pattern is `Mapper<? extends CharSequence>`.

Once we know the type of the record pattern, we can find its component types, which are matched

against the nested patterns. Pattern variables `in` and `out` both have type `CharSequence`.

DRAFT 21-internal-adhoc.gbierman.20230509