Java language feature history

Java language changes

- JDK 1.0 – 1996
  - Initial release
- JDK 1.1 – 1997
  - Inner classes
- JDK 1.2, 1.3 – 1998, 2000
  - No changes at the language level
- JDK 1.4 – 2002
  - Assertions (minor change)
- JDK 1.5 (Java 5) – 2005
  - Biggest changes to the language since the initial release
- JDK 1.6 (Java 6) – 2006
  - No changes at the language level

Java language changes

- JDK 1.7 (Java 7) – 2011
  - Several minor language changes
- JDK 1.8 (Java 8) – 2014
  - Several major language changes related to functions/methods
- JDK 1.9 (Java 9) – September 2017
  - Minor language changes
- JDK 1.10 (Java 10) – March 2018
  - Minor language changes
- JDK 1.11 (Java 11) – September 2018
  - Minor language changes
- JDK 1.12–1.15 (Java 12–15) – March/September 2019–2020
  - Minor language changes

Java 5 language improvements

- Enumerated types
- Generics
- Autoboxing/Unboxing
- Varargs
- Enhanced *for* loop
- Static imports
- Annotations
- Covariant return types

Java 7 language improvements

- Numeric literals
- Strings in switch statements
- Type inference for generic instances
- Try-with-resources statement
- Catching multiple exception types

Java 8 language improvements

- Lambda expressions
- Method references
- Functional interfaces
- Default methods

Java 9 language improvements

- Modules
- Private interface methods

Java 10 & 11 language improvements

- Local-variable type inference

Java 12, 13, 14 & 15 language improvements

- Standard feature: switch expressions
- Preview features: instanceof, records, text blocks, sealed classes

Java 5 language improvements

- Enumerated types
- Generics
- Autoboxing/Unboxing
- Varargs
- Enhanced *for* loop
- Static imports
- Annotations
- Covariant return types

Enumerated types

- Enumerated types should be used on data sets where all possible values are known at compile-time
- Prior to Java 1.5, enumerated types were simulated using a set of ordinal constants (e.g. integers):

```
class Apples {
  public static final int GOLDEN_DELICIOUS = 0;
  public static final int RED_DELICITIOUS  = 1;
  public static final int MUTSU            = 2;
}
```

- Disadvantages of this approach:
  - No compile-time type checking
  - Unable to iterate through all values
  - Difficult to associate arbitrary data to enumerated values

Enumerations in Java 1.5

- An *enum* type: a reference type whose *fields* consist of a fixed set of comma-separated constants:

```
enum Apples {
  GOLDEN_DELICIOUS, RED_DELICIOUS, MUTSU
}
```

- Besides fields, enumerations can have properties, constructors, methods, inner classes, etc.
- Each field in an enumeration can have its own method implementation, expressing different behavior
- There is also a (limited) support for inheritance

Java 5 language improvements

- Enumerated types
- Generics
- Autoboxing/Unboxing
- Varargs
- Enhanced *for* loop
- Static imports
- Annotations
- Covariant return types

Generics

- Generics allow developers to abstract over *types*
  - Classes, interfaces, and methods can be *parameterized* by types
- The effect of using generics is *type-safe code:*
  - If the code compiles without errors or warnings, then it will certainly not throw a typecasting exception at run-time
- The most common usage of generics is with collections of objects
- Generics make code easier to read
  - Once you get used to the syntax

Generics example

- Example of a parameterized *Stack* class that accepts any reference type:

```
class Stack<E> {
  void push(E element) { ... }
  void pop() { ... }
  E top() { ... }
}
```

- The class is used by substituting the *type parameter E* with the concrete *type argument:*

```
// creating a stack of Strings
Stack<String> stack = new Stack<String>();
```

- *More on generics in a dedicated topic*

Java 5 language improvements

- Enumerated types
- Generics
- Autoboxing/Unboxing
- Varargs
- Enhanced *for* loop
- Static imports
- Annotations
- Formatting
- Covariant return types

Autoboxing/Unboxing

- *Box*
  - Instance of the wrapper class that holds the value of a primitive type
- Example wrapper classes: *Integer, Float, Boolean*, etc.
- *Boxing*
  - Creating a box for a primitive value
- *Unboxing*
  - Removing the primitive value from a box

Manual boxing / unboxing

- Instances of primitive types are *not* objects. A reference type is used to *wrap* the primitive value into an object
- In some cases, such as Collections, reference types must be used
- Before Java 1.5, a primitive value had to be manually wrapped into a reference type:

```
int i = 5;
LinkedList list = new LinkedList();
list.add(new Integer(i)); // boxing
```

- To get the value, the reference type had to be unboxed using a dedicated method:

```
Integer n = list.get(0);
int i = n.intValue(); // unboxing
```

Automatic boxing / unboxing

- Since Java 1.5, the compiler performs automatic boxing and unboxing:

```
Integer n = 5; // auto-boxing
int i = n;     // auto-unboxing

int j = 10;
LinkedList list = new LinkedList();
list.add(j);          // auto-boxing
int k = list.get(0); // auto-unboxing
```

- Control statements, such as *if*, *while*, and *do*, can now also use *Boolean*, instead of requiring *boolean*
- Clear advantages: less code to write, easier to read

Disadvantages of auto-(un)boxing

- Be careful when performing a comparison:

```
Integer a = new Integer(7);
int b = 7;
Integer c = new Integer(7);
if (a == b) ... // true
if (a == c) ... // false, a and c are references!
// for comparing references, use:
if (a.equals(c)) ...
```

- Performance issues: the following code auto-boxes *i* in every iteration, resulting in lowered performance:

```
LinkedList list = new LinkedList();
for (int i = 0; i < 1000; i++)
  list.add(i); // boxing hidden, but there
```

- Be careful not to unbox a *null*; an exception will be thrown

Conclusion on auto-(un)boxing

- Automatic boxing and unboxing blur the distinction between primitive and reference types, but they do not eliminate it
- An *Integer* is **not** a substitute for an *int*!
- Avoid using these feature in performance-sensitive code; be aware of them, and apply only when justified

Java 5 language improvements

- Enumerated types
- Generics
- Autoboxing/Unboxing
- Varargs
- Enhanced *for* loop
- Static imports
- Annotations
- Covariant return types

Varargs

- The feature that allows for methods to be defined using variable number of arguments (zero or more)
- All arguments have to be of the same type
- The number of arguments need not be predetermined
- Before Java 1.5, variable number of parameters was simulated using an array:

```
// method definition
void oldPrintAll(int k, String[] strings) {
  System.out.println(k);
  for (int i = 0; i < strings.length; i++)
    System.out.println(strings[i]);
}

// when invoking, the array needs to be created
String[] strings = { "foo", "bar" };
oldPrintAll(7, strings);
```

Varargs

- Vararg parameter is denoted by ellipsis ("…")
- Inside a method it is handled as a regular array
- However, the method can be called with any number of parameters, without the need of putting them in an array:

```
// method definition
void newPrintAll(int k, String... strings) {
  System.out.println(k);
  for (int i = 0; i < strings.length; i++)
    System.out.println(strings[i]);
}

// invoking the new method - no need for an array!
newPrintAll(6, "foo", "bar");
// the vararg can also receive zero parameters
newPrintAll(6);
```

Vararg rules

- The vararg can only be the very last parameter in a method definition
- In general, avoid using varargs, especially when overloading is required, as it might be difficult for the reader to figure out which method is being called:

```
void someMethod(String str) {
  System.out.println("Inside the first method");
}

void someMethod(String a, String... b) {
  System.out.println("Inside the second method");
}

void mainMethod() {
  // which one of the above gets called?!
  someMethod("Hello");
}
```

Java 5 language improvements

- Enumerated types
- Generics
- Autoboxing/Unboxing
- Varargs
- Enhanced *for* loop
- Static imports
- Annotations
- Covariant return types

Enhanced *for* loop

- The new *for* loop introduced in Java 1.5 enables easier iteration through a collection or an array, without the need for defining an iterator or an indexing variable
- Especially when using the *Iterator* for collections, this results in a shorter, easier-to-read code
- General syntax for a collection that has an *Iterator*:
  - *for* (type variable : collection)
- Works for arrays as well:
  - *for* (type variable : array)

Enhanced *for* loop – using with a collection

```
// the old way of iterating through a collection
void print(Collection<Integer> c)
{
  for (Iterator<Integer> i = c.iterator(); i.hasNext();)
    System.out.println(i.next());
}

// the enhanced for loop: much better code
void print(Collection<Integer> c)
{
  for (int n : c) // automatic unboxing applies
    System.out.println(n);
}
```

Enhanced *for* loop – using with an array

```
// the old way
void print(int[] numbers) {
  for (int i = 0; i < numbers.length; i++)
    System.out.println(numbers[i]);
}

// the new way
void print(int[] numbers) {
  for (int n : numbers)
    System.out.println(n);
}
```

- Of course, it cannot be used if you need to know the exact index of the element you're dealing with, for example:

```
for (int i = 0; i < array.length; i++)
  System.out.println("Element " + i + " is " + array[i]);
```

Java 5 language improvements

- Enumerated types
- Generics
- Autoboxing/Unboxing
- Varargs
- Enhanced *for* loop
- Static imports
- Annotations
- Covariant return types

Static imports

- Problem: Having to fully qualify every static member referenced from external classes. For example:

```
double r = Math.cos(Math.PI * theta);
System.out.println(r);
```

- Solution: the new import syntax:

```
import static java.lang.System.out;
import static java.lang.Math.cos;
import static java.lang.Math.PI;
```

- The imported static members can now be used without qualifications:

```
double r = cos(PI * theta);
out.println(r);
```

- The wildcard “*” can still be used:

```
import static java.lang.Math.*;
```

Java 5 language improvements

- Enumerated types
- Generics
- Autoboxing/Unboxing
- Varargs
- Enhanced *for* loop
- Static imports
- Annotations
- Covariant return types

Annotations

- Language constructs that assign additional *semantics* to source code elements, such as classes, methods, and fields
- Annotations can be processed:
  - At compile time, by the compiler itself
  - At compile time, by an external software tool that, for example, outputs XML
  - At run-time, by an external tool or library
- General syntax:
  - @annotationName (optionalAruments) *elementThatIsBeingAnnotated*
- Developers can write their own annotations, but this feature is rarely used

Built-in annotations

- @Override
  - Informs the compiler that the annotated method is meant to override a method in the super-type
- @Deprecated
  - Discourages the use of the annotated element, usually because there is a better alternative (e.g. indicates an old element of the library that remains solely for backward-compatibility)
- @SuppressWarnings({set_of_warnings})
  - Informs the compiler to ignore the given set of warnings when processing the annotated element

Using @Override

- In order to avoid hard-to-detect errors, always use the @Override annotation when overriding a method:

```
class Person {
  private String name;

  public Person(String name) {
    this.name = name;
  }

  public String toString() {
    return "Name of the person: " + name;
  }
}

public class Annotations {
  public static void main(String[] args) {
    Person p = new Person("John");
    System.out.println(p); // outputs Person@9304b1. Why?
  }
}
```

Using @Override

```
class Person {
  private String name;

  public Person(String name) {
    this.name = name;
  }

  @Override
  public String toStringq() { // compiler error
    return "Name of the person: " + name;
  }
}

public class Annotations {
  public static void main(String[] args) {
    Person p = new Person("John");
    System.out.println(p);
  }
}
```

- Compiler generates an error: there is no “toStringq” method in the super-class (in this case, *Object*)

Java 5 language improvements

- Enumerated types
- Generics
- Autoboxing/Unboxing
- Varargs
- Enhanced *for* loop
- Static imports
- Annotations
- Covariant return types

Covariant return types

- It is now legal for an overriding method's return type to be a subclass of the overridden method's return type
- This allows the overriding method to provide more information about the returned object and eliminates the need for casting in the client:

```
class A {
  public A makeCopy() {  return new A(); }
}

class B extends A {
  @Override
  public B makeCopy() { // this is now legal
    return new B();
  }
}
```

Java 7 language improvements

- Numeric literals
- Strings in switch statements
- Type inference for generic instances
- Try-with-resources statement
- Catching multiple exception types

Numeric literals

- Binary literals:

```
// An 8-bit 'byte' value:
byte aByte = (byte)0b00100001;
// A 16-bit 'short' value:
short aShort = (short)0b1010000101000101;
// Some 32-bit 'int' values:
int anInt1 = 0b10100001010001011010000101000101;
int anInt2 = 0B101; // The B can be upper or lower case
```

- Underscores allowed between digits in numeric literals, for grouping digits and improving readability of code:

```
long creditCardNumber = 1234_5678_9012_3456L;
float pi = 3.14_15F;
long hexBytes = 0xFF_EC_DE_5E;
long maxLong = 0x7fff_ffff_ffff_ffffL;
long bytes = 0b11010010_01101001_10010100_10010010;
```

Java 7 language improvements

- Numeric literals
- Strings in switch statements
- Type inference for generic instances
- Try-with-resources statement
- Catching multiple exception types

Strings in switch statements

- A String object can be used in the expression of a switch statement. Strings are compared as if the String.equals method was used. Example:

```
public String getTypeOfDay(String dayOfWeek) {
  String typeOfDay;
  switch (dayOfWeek) {
    case "Monday":
      typeOfDay = "Start of work week";
      break;
    case "Tuesday": case "Wednesday": case "Thursday":
      typeOfDay = "Midweek";
      break;
    case "Friday":
      typeOfDay = "End of work week";
      break;
    case "Saturday": case "Sunday":
      typeOfDay = "Weekend";
      break;
    default:
      throw new IllegalArgumentException("Invalid day of the week: " + dayOfWeek);
  }
  return typeOfDay;
}
```

Java 7 language improvements

- Numeric literals
- Strings in switch statements
- Type inference for generic instances
- Try-with-resources statement
- Catching multiple exception types

Type inference for generic instances

- It is possible to replace the type arguments required to invoke the constructor of a generic class with an empty set of type parameters (<>) as long as the compiler can infer the type arguments from the context:

```
// "Regular" instance creation:
Map<String, List<String>> myMap = new HashMap<String, List<String>>();

// Instance creation with type inference (<> obligatory):
Map<String, List<String>> myMap = new HashMap<>();

// Unchecked conversion warning:
Map<String, List<String>> myMap = new HashMap();
```

Java 7 language improvements

- Numeric literals
- Strings in switch statements
- Type inference for generic instances
- Try-with-resources statement
- Catching multiple exception types

Try-with-resources statement

- The try-with-resources statement is a try statement that declares one or more resources
- A *resource* is as an object that must be closed after the program is finished with it
- The try-with-resources statement ensures that each resource is closed at the end of the statement
- Any object that implements java.lang.AutoCloseable, which includes all objects that implement java.io.Closeable, can be used as a resource

Try-with-resources statement

- Instead of

```
static String readFirstLineFromFile(String path) throws IOException {
  BufferedReader br = new BufferedReader(new FileReader(path));
  try {
    return br.readLine();
  } finally {
    if (br != null) br.close();
  }
}
```

we can now write

```
static String readFirstLineFromFile(String path) throws IOException {
  try (BufferedReader br = new BufferedReader(new FileReader(path))) {
    return br.readLine();
  }
}
```

- Multiple resources can be specified, separated by ;

Java 7 language improvements

- Numeric literals
- Strings in switch statements
- Type inference for generic instances
- Try-with-resources statement
- Catching multiple exception types

Catching multiple exception types

- One catch block can handle more than one type of exception
- This feature can reduce code duplication and lessen the temptation to catch an overly broad exception
- Example:

```
catch (IOException ex) {
  logger.log(ex);
  throw ex;
}
catch (SQLException ex) {
  logger.log(ex);
  throw ex;
}
```

can be written as

```
catch (IOException|SQLException ex) {
  logger.log(ex);
  throw ex;
}
```

Java 8 language improvements

- Lambda expressions
- Method references
- Functional interfaces
- Default methods

Lambda expressions

- A *lambda expression* is an anonymous function/method

- It consists of:
  - Zero or more parameters: the parameters can be written with or without the type
  - The lambda operator ->
  - A body: if the body contains only one statement, the braces are not necessary

Examples

```
() -> System.out.println("Hello, World!");

(int x) -> x % 2 == 0

(x) -> x % 2 == 0

(a, b) -> {
  int sum = a + b;
  System.out.println("Sum: " + sum);
}
```

Using lambda expressions

- In Java, lambda expressions are most commonly used for:
  - Processing arrays/collections
  - As a replacement for anonymous inner classes
- Java 8 includes then new *Stream API* with a number of methods that accept lambda expressions
  - `forEach()`, `filter()`, `map()`
- A collection is easily transformed into a stream by calling its `stream()` method
  - For arrays, there's the static `Arrays.stream()` method
- *More on lambda expressions in a dedicated topic*

Java 8 language improvements

- Lambda expressions
- Method references
- Functional interfaces
- Default methods

Method references

- With *method references*, you can avoid writing a lambda expression that is already implemented in an existing method
  - Instead of providing a lambda expression, you provide a reference to the method
- For example, instead of writing

```
list.stream().forEach(str -> System.out.println(str));
```

- You can write:

```
list.stream().forEach(System.out::println);
```

- The given method is now applied to every element of the list

How to reference a method

- You can reference:
  - A static method
    - `ClassName::method`
  - Method of an object
    - `object::method`
  - Non-static method of a reference type
    - E.g. `String::equals`
  - A constructor
    - `ClassName::new`

Java 8 language improvements

- Lambda expressions
- Method references
- Functional interfaces
- Default methods

Functional interfaces

- A *functional interface* is an interface with only one method
- It allows you to write methods that accept lambda expressions
- Example functional interface:

```
@FunctionalInterface
interface Predicate {
  void apply(String str);
}
```

- The annotation is optional: it tells the compiler that this is a functional interface; it will complain if you add more methods

Using functional interfaces

- A method that accepts a functional interface and a list of strings, and applies the interface method to each element:

```
void process(Predicate p, List<String> list) {
  for (String str : list)
    p.apply(str);
}
```

- The method can be called as:

```
process(str -> System.out.println(str), list);
```

- In the call, you practically provide an implementation for the method `apply`

Java 8 language improvements

- Lambda expressions
- Method references
- Functional interfaces
- Default methods

Default methods

- As of Java 8, interfaces can have implemented methods:

```
interface Person {
  String getName();

  default boolean sameName(Person other) {
    return getName().equals(other.getName());
  }
}
```

- This is useful if you need to add methods to an existing interface in a large project
  - A method with a default implementation can be added to an interface without the need to change all implementing classes
- Java library developers used this feature to extend standard interfaces with support for lambda expressions

Java 9 language improvements

- Modules
- Private interface methods

Modules

- Before Java 9, there were great problems with:
  - Encapsulation (e.g. accessing private fields was easy)
  - Reliable configuration (e.g. impostors in CLASSPATH)
- In Java 9, a JAR file can contain `module-info.java` like this:

```
module mymodule {
    exports mypackage;
    requires someothermodule;
}
```

- Only the contents of `mypackage` are visible to other modules
- `mymodule` can access only the contents from the modules it `requires`
- MODULE-PATH now exists in addition to CLASSPATH
- "Unnamed module" provides support for old behavior

Java 9 language improvements

- Modules
- Private interface methods

Private interface methods

- Since default methods were introduced to Java 8, a common refactoring became a problem: Moving repeating code to a new private helper method

- In Java 9, this is now possible:

```
public interface MyInterface {
  void normalMethod();
  default void defaultMethod() { init(); }
  default void anotherDefaultMethod() { init(); }
  private void init() {
    System.out.println("Initializing");
  }
}
```

Java 10 & 11 language improvements

- Local-variable type inference

Local-variable type inference

- Possibility to avoid explicit type declaration of local variables
- Restricted to:
  - Local variables with initializers
  - Indexes in the enhanced for-loop
  - Local variables declared in a traditional for-loop

- From Java 10 & 11, this is now possible:

```
// infers ArrayList<String>
var list = new ArrayList<String>();

// infers Stream<String>
var stream = list.stream();
```

Java 12, 13, 14 & 15 language improvements

- Standard feature: switch expressions
- Preview features: instanceof, records, text blocks, sealed classes

Switch expressions (Standard in Java 14)

- What the ? : is in relation to the if statement, switch expressions are to the switch statement
- From Java 14, this is now possible:

```
String result = switch (day) {
  case "M", "W", "F" -> "MWF";
  case "T", "TH", "S" -> "TTS";
  default -> {
    if (day.isEmpty())
      yield "Please insert a valid day.";
    else
      yield "Looks like a Sunday.";
  }
};
```

Java 12, 13, 14 & 15 language improvements

- Standard feature: switch expressions
- Preview features: instanceof, records, text blocks, sealed classes

Preview features

- Preview features (as well as Experimental and Incubator) are not available without using special compiler flags / modules
- Pattern matching for instanceof:

```
if (o instanceof String) {      if (o instanceof String s) {
  String s = (String)o; ...}        ...}
```

- Records: group related fields together as a single immutable data item

```
record Author (String name, String topic) {}
```

- Text blocks:

```
String s1 = """                 String s2 = "line 1\nline 2";
            line 1
            line 1
            """;
```

Preview features

- Sealed classes:

```
public abstract sealed class Person
    permits Employee, Manager {
    ...
}
```

- Any class that extends a sealed class must itself be declared sealed, non-sealed, or final:

```
public final class Employee extends Person { ... }

public non-sealed class Manager extends Person { ... }
```

- This allows the compiler to determine whether a class hierarchy is finite, and use that information to check, e.g., `if-else` statements that employ `instanceof`