
Java Annotations

In this tutorial, we will learn what annotations are, different Java annotations and how to use them with the help of examples.

Java annotations are metadata (data about data) for our program source code.

They provide additional information about the program to the compiler but are not part of the program itself. These annotations do not affect the execution of the compiled program.

Annotations start with `@`. Its syntax is:

```
@AnnotationName
```

Let's take an example of `@Override` annotation.

The `@Override` annotation specifies that the method that has been marked with this annotation overrides the method of the superclass with the same method name, return type, and parameter list.

It is not mandatory to use `@Override` when overriding a method. However, if we use it, the compiler gives an error if something is wrong (such as wrong parameter type) while overriding the method.

Example 1: @Override Annotation Example

```
class Animal {

  public void displayInfo() {

    System.out.println("I am an animal.");

  }

}

class Dog extends Animal {

  @Override

  public void displayInfo() {

    System.out.println("I am a dog.");

  }
```

```
}

class Main {

  public static void main(String[] args) {

    Dog d1 = new Dog();

    d1.displayInfo();

  }

}
```

Output

```
I am a dog.
```

In this example, the method `displayInfo()` is present in both the superclass `Animal` and subclass `Dog`. When this method is called, the method of the subclass is called instead of the method in the superclass.

Annotation formats

Annotations may also include elements (members/attributes/parameters).

1. Marker Annotations

Marker annotations do not contain members/elements. It is only used for marking a declaration.

Its syntax is:

```
@AnnotationName()
```

Since these annotations do not contain elements, parentheses can be excluded. For example,

```
@Override
```

---

2. Single element Annotations

A single element annotation contains only one element.

Its syntax is:

```
@AnnotationName(elementName = "elementValue")
```

If there is only one element, it is a convention to name that element as `value`.

```
@AnnotationName(value = "elementValue")
```

In this case, the element name can be excluded as well. The element name will be `value` by default.

```
@AnnotationName("elementValue")
```

---

3. Multiple element Annotations

These annotations contain multiple elements separated by commas.

Its syntax is:

```
@AnnotationName(element1 = "value1", element2 = "value2")
```

Annotation placement

Any declaration can be marked with annotation by placing it above that declaration. As of Java 8, annotations can also be placed before a type.

1. Above declarations

As mentioned above, Java annotations can be placed above class, method, interface, field, and other program element declarations.

Example 2: @SuppressWarnings Annotation Example

```
import java.util.*;

class Main {
```

```
  @SuppressWarnings("unchecked")
  static void wordsList() {
    ArrayList wordList = new ArrayList<>();
// This causes an unchecked warning
    wordList.add("programiz");

    System.out.println("Word list => " + wordList);
  }

  public static void main(String args[]) {
    wordsList();
  }
}
```

Output

```
Word list => [programiz]
```

If the above program is compiled without using the `@SuppressWarnings("unchecked")` annotation, the compiler will still compile the program but it will give warnings like:

```
Main.java uses unchecked or unsafe operations.

Word list => [programiz]
```

We are getting the warning

```
Main.java uses unchecked or unsafe operations
```

because of the following statement.

```
ArrayList wordList = new ArrayList<>();
```

This is because we haven't defined the generic type of the array list. We can fix this warning by specifying generics inside angle brackets `<>`.

```
ArrayList<String> wordList = new ArrayList<>();
```

---

2. Type annotations

Before Java 8, annotations could be applied to declarations only. Now, type annotations can be used as well. This means that we can place annotations wherever we use a type.

Constructor invocations

```
new @Readonly ArrayList<>()
```

Type definitions

```
@NonNull String str;
```

This declaration specifies non-null variable `str` of type `String` to avoid `NullPointerException`.

```
@NonNull List<String> newList;
```

This declaration specifies a non-null list of type `String`.

```
List<@NonNull String> newList;
```

This declaration specifies a list of non-null values of type `String`.

Type casts

```
newStr = (@NonNull String) str;
```

extends and implements clause

```
class Warning extends @Localized Message
```

throws clause

```
public String readMethod() throws @Localized IOException
```

Type annotations enable Java code to be analyzed better and provide even stronger type checks.

Types of Annotations

1. Predefined annotations

1. `@Deprecated`
2. `@Override`
3. `@SuppressWarnings`
4. `@SafeVarargs`
5. `@FunctionalInterface`

2. Meta-annotations

1. `@Retention`
2. `@Documented`
3. `@Target`
4. `@Inherited`
5. `@Repeatable`

3. Custom annotations

These annotation types are described in detail in the Java Annotation Types tutorial.

Use of Annotations

- **Compiler instructions** - Annotations can be used for giving instructions to the compiler, detect errors or suppress warnings. The built-in annotations `@Deprecated`, `@Override`, `@SuppressWarnings` are used for these purposes.
- **Compile-time instructions** - Compile-time instructions provided by these annotations help the software build tools to generate code, XML files and many more.
- **Runtime instructions** - Some annotations can be defined to give instructions to the program at runtime. These annotations are accessed using Java Reflection.

Java Annotation Types

In this tutorial, we will learn about different types of Java annotation with the help of examples.

Java annotations are metadata (data about data) for our program source code. There are several predefined annotations provided by the Java SE. Moreover, we can also create custom annotations as per our needs.

If you do not know what annotations are, visit the Java annotations tutorial.

These annotations can be categorized as:

1. **Predefined annotations**

- `@Deprecated`
- `@Override`
- `@SuppressWarnings`
- `@SafeVarargs`
- `@FunctionalInterface`

2. **Custom annotations**

3. **Meta-annotations**

- `@Retention`
- `@Documented`
- `@Target`
- `@Inherited`
- `@Repeatable`

Predefined Annotation Types

1. @Deprecated

The `@Deprecated` annotation is a marker annotation that indicates the element (class, method, field, etc) is deprecated and has been replaced by a newer element.

Its syntax is:

```
@DeprecatedaccessModifier returnType deprecatedMethodName() { ... }
```

When a program uses the element that has been declared deprecated, the compiler generates a warning.

We use Javadoc `@deprecated` tag for documenting the deprecated element.

```
/**
 * @deprecated
 * why it was deprecated
 */@DeprecatedaccessModifier returnType deprecatedMethodName() { ... }
```

Example 1: @Deprecated annotation example

```
class Main {
  /**
   * @deprecated
   * This method is deprecated and has been replaced by newMethod()
   */
  @Deprecated
  public static void deprecatedMethod() {
    System.out.println("Deprecated method");
  }

  public static void main(String args[]) {
    deprecatedMethod();
  }
}
```

Output

```
Deprecated method
```

---

2. @Override

The `@Override` annotation specifies that a method of a subclass overrides the method of the superclass with the same method name, return type, and parameter list.

It is not mandatory to use `@Override` when overriding a method. However, if we use it, the compiler gives an error if something is wrong (such as wrong parameter type) while overriding the method.

Example 2: @Override annotation example

```
class Animal {


  // overridden method
  public void display(){
    System.out.println("I am an animal");
  }
}
class Dog extends Animal {


  // overriding method
  @Override
  public void display(){
    System.out.println("I am a dog");
  }


  public void printMessage(){
    display();
  }
}
class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
    dog1.printMessage();
  }
```

```
}
```

Output

```
I am a dog
```

In this example, by making an object `dog1` of `Dog` class, we can call its method `printMessage()` which then executes the `display()` statement.

Since `display()` is defined in both the classes, the method of subclass `Dog` overrides the method of superclass `Animal`. Hence, the `display()` of the subclass is called.

---

3. @SuppressWarnings

As the name suggests, the `@SuppressWarnings` annotation instructs the compiler to suppress warnings that are generated while the program executes.

We can specify the type of warnings to be suppressed. The warnings that can be suppressed are compiler-specific but there are two categories of warnings: **deprecation** and **unchecked**.

To suppress a particular category of warning, we use:

```
@SuppressWarnings("warningCategory")
```

For example,

```
@SuppressWarnings("deprecated")
```

To suppress multiple categories of warnings, we use:

```
@SuppressWarnings({"warningCategory1", "warningCategory2"})
```

For example,

```
@SuppressWarnings({"deprecated", "unchecked"})
```

Category `deprecated` instructs the compiler to suppress warnings when we use a deprecated element.

Category `unchecked` instructs the compiler to suppress warnings when we use raw types.

And, undefined warnings are ignored. For example,

```
@SuppressWarnings("someundefinedwarning")
```

Example 3: @SuppressWarnings annotation example

```
class Main {

  @Deprecated

  public static void deprecatedMethod() {

    System.out.println("Deprecated method");

  }


  @SuppressWarnings("deprecated")

  public static void main(String args[]) {

    Main depObj = new Main();

    depObj. deprecatedMethod();

  }

}
```

Output

```
Deprecated method
```

Here, `deprecatedMethod()` has been marked as deprecated and will give compiler warnings when used. By using the `@SuppressWarnings("deprecated")` annotation, we can avoid compiler warnings.

---

4. @SafeVarargs

The `@SafeVarargs` annotation asserts that the annotated method or constructor does not perform unsafe operations on its varargs (variable number of arguments).

We can only use this annotation on methods or constructors that cannot be overridden. This is because the methods that override them might perform unsafe operations.

Before Java 9, we could use this annotation only on final or static methods because they cannot be overridden. We can now use this annotation for private methods as well.

Example 4: @SafeVarargs annotation example

```
import java.util.*;

class Main {


  private void displayList(List<String>... lists) {

    for (List<String> list : lists) {

      System.out.println(list);

    }

  }


  public static void main(String args[]) {

    Main obj = new Main();


    List<String> universityList = Arrays.asList("Tribhuvan University", "Kathmandu University");

    obj.displayList(universityList);


    List<String> programmingLanguages = Arrays.asList("Java", "C");

    obj.displayList(universityList, programmingLanguages);

  }

}
```

Warnings

```
Type safety: Potential heap pollution via varargs parameter lists

Type safety: A generic array of List<String> is created for a varargs

 parameter
```

Output

```
Note: Main.java uses unchecked or unsafe operations.
```

```
[Tribhuvan University, Kathmandu University]

[Tribhuvan University, Kathmandu University]

[Java, C]
```

Here, `List ... lists` specifies a variable-length argument of type `List`. This means that the method `displayList()` can have zero or more arguments.

The above program compiles without errors but gives warnings when `@SafeVarargs` annotation isn't used.

When we use `@SafeVarargs` annotation in the above example,

```
@SafeVarargs

 private void displayList(List<String>... lists) { ... }
```

We get the same output but without any warnings. Unchecked warnings are also suppressed when we use this annotation.

---

5. @FunctionalInterface

Java 8 first introduced this `@FunctionalInterface` annotation. This annotation indicates that the type declaration on which it is used is a functional interface. A functional interface can have only one abstract method.

Example 5: @FunctionalInterface annotation example

```
@FunctionalInterfacepublic interface MyFuncInterface{

  public void firstMethod(); // this is an abstract method

}
```

If we add another abstract method, let's say

```
@FunctionalInterfacepublic interface MyFuncInterface{

  public void firstMethod(); // this is an abstract method

  public void secondMethod(); // this throws compile error

}
```

Now, when we run the program, we will get the following warning:

```
Unexpected @FunctionalInterface annotation

@FunctionalInterface ^ MyFuncInterface is not a functional interface

multiple non-overriding abstract methods found in interface MyFuncInterface
```

It is not mandatory to use `@FunctionalInterface` annotation. The compiler will consider any interface that meets the functional interface definition as a functional interface.

We use this annotation to make sure that the functional interface has only one abstract method.

However, it can have any number of default and static methods because they have an implementation.

```
@FunctionalInterfacepublic interface MyFuncInterface{

  public void firstMethod(); // this is an abstract method

  default void secondMethod() { ... }

  default void thirdMethod() { ... }

}
```

Custom Annotations

It is also possible to create our own custom annotations.

Its syntax is:

```
[Access Specifier] @interface<AnnotationName> {

  DataType <Method Name>() [default value];

}
```

Here is what you need to know about custom annotation:

- Annotations can be created by using `@interface` followed by the annotation name.
- The annotation can have elements that look like methods but they do not have an implementation.

- The default value is optional. The parameters cannot have a null value.
- The return type of the method can be primitive, enum, string, class name or array of these types.

Example 6: Custom annotation example

```
@interface MyCustomAnnotation {

  String value() default "default value";

}

class Main {

  @MyCustomAnnotation(value = "programiz")

  public void method1() {

    System.out.println("Test method 1");

  }


  public static void main(String[] args) throws Exception {

    Main obj = new Main();

    obj.method1();

  }

}
```

Output

```
Test method 1
```

Meta Annotations

Meta-annotations are annotations that are applied to other annotations.

1. @Retention

The `@Retention` annotation specifies the level up to which the annotation will be available.

Its syntax is:

```
@Retention(RetentionPolicy)
```

There are 3 types of retention policies:

- **RetentionPolicy.SOURCE** - The annotation is available only at the source level and is ignored by the compiler.
- **RetentionPolicy.CLASS** - The annotation is available to the compiler at compile-time, but is ignored by the Java Virtual Machine (JVM).
- **RetentionPolicy.RUNTIME** - The annotation is available to the JVM.

For example,

```
@Retention(RetentionPolicy.RUNTIME)public @interface MyCustomAnnotation{ ... }
```

---

2. @Documented

By default, custom annotations are not included in the official Java documentation. To include our annotation in the Javadoc documentation, we use the `@Documented` annotation.

For example,

```
@Documentedpublic @interface MyCustomAnnotation{ ... }
```

---

3. @Target

We can restrict an annotation to be applied to specific targets using the `@Target` annotation.

Its syntax is:

```
@Target(ElementType)
```

The `ElementType` can have one of the following types:

Element Type	Target

`ElementType.ANNOTATION_TYPE`	Annotation type
`ElementType.CONSTRUCTOR`	Constructors
`ElementType.FIELD`	Fields
`ElementType.LOCAL_VARIABLE`	Local variables
`ElementType.METHOD`	Methods
`ElementType.PACKAGE`	Package
`ElementType.PARAMETER`	Parameter
`ElementType.TYPE`	Any element of class

For example,

```
@Target(ElementType.METHOD)public @interface MyCustomAnnotation{ ... }
```

In this example, we have restricted the use of this annotation to methods only.

> **Note:** If the target type is not defined, the annotation can be used for any element.

---

4. @Inherited

By default, an annotation type cannot be inherited from a superclass. However, if we need to inherit an annotation from a superclass to a subclass, we use the `@Inherited` annotation.

Its syntax is:

```
@Inherited
```

For example,

```
@Inheritedpublic @interface MyCustomAnnotation { ... }

@MyCustomAnnotationpublic class ParentClass{ ... }

public class ChildClass extends ParentClass { ... }
```

---

5. @Repeatable

An annotation that has been marked by `@Repeatable` can be applied multiple times to the same declaration.

```
@Repeatable(Universities.class)public @interface University {

  String name();

}
```

The value defined in the `@Repeatable` annotation is the container annotation. The container annotation has a variable `value` of array type of the above repeatable annotation.
Here, `Universities` are the containing annotation type.

```
public @interface Universities {

  University[] value();

}
```

Now, the `@University` annotation can be used multiple times on the same declaration.

```
@University(name = "TU")@University(name = "KU")private String uniName;
```

If we need to retrieve the annotation data, we can use the Reflection API.

To retrieve annotation values, we use `getAnnotationsByType()` or `getAnnotations()` method defined in the Reflection API.

Java Logging

In this tutorial, we will learn about Java Logging and its various components with the help of examples.

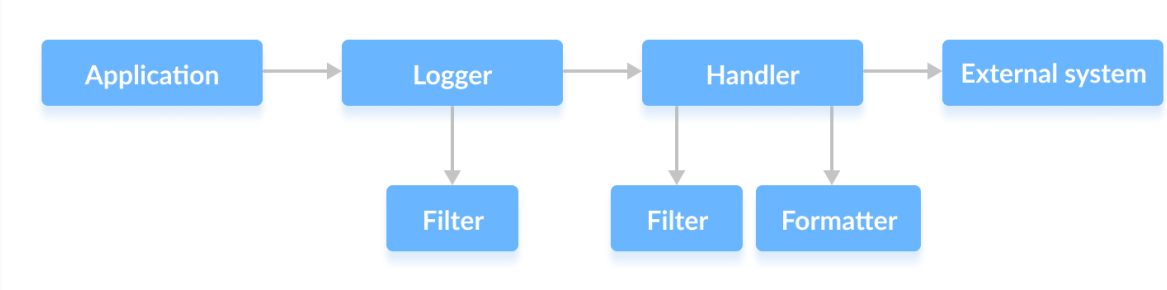
Java allows us to create and capture log messages and files through the process of logging.

In Java, logging requires frameworks and APIs. Java has a built-in logging framework in the `java.util.logging` package.

We can also use third-party frameworks like Log4j, Logback, and many more for logging purposes.

Java Logging Components

The figure below represents the core components and the flow of control of the Java Logging API (`java.util.logging`).



Java Logging

1. Logger

The `Logger` class provides methods for logging. We can instantiate objects from the `Logger` class and call its methods for logging purposes.

Let's take an example.

```
Logger logger = Logger.getLogger("newLoggerName");
```

The `getLogger()` method of the `Logger` class is used to find or create a new `Logger`. The string argument defines the name of the logger.

Here, this creates a new `Logger` object or returns an existing `Logger` with the same name.

It is a convention to define a `Logger` after the current class using `class.getName()`.

```
Logger logger = Logger.getLogger(MyClass.class.getName());
```

> **Note:** This method will throw `NullPointerException` if the passed name is `null`.

Each `Logger` has a level that determines the importance of the log message. There are 7 basic log levels:

Log Level (in descending order)	Use
SEVERE	serious failure
WARNING	warning message, a potential problem
INFO	general runtime information
CONFIG	configuration information
FINE	general developer information (tracing messages)
FINER	detailed developer information (tracing messages)
FINEST	highly detailed developer information(tracing messages)
OFF	turn off logging for all levels (capture nothing)
ALL	turn on logging for all levels (capture everything)

Each log level has an integer value that determines their severity except for two special log levels `OFF` and `ALL`.

Logging the message

By default, the top three log levels are always logged. To set a different level, we can use the following code:

```
logger.setLevel(Level.LogLevel);

// example
```

```
logger.setLevel(Level.FINE);
```

In this example, only level `FINE` and levels above it are set to be logged. All other log messages are dropped.

Now to log a message, we use the `log()` method.

```
logger.log(Level.LogLevel, "log message");

// example

logger.log(Level.INFO, "This is INFO log level message");
```

There are shorthand methods for logging at desired levels.

```
logger.info( "This is INFO log level message");

logger.warning( "This is WARNING log level message");
```

All log requests that have passed the set log level are then forwarded to the **LogRecord**.

> **Note:** If a logger's level is set to `null`, its level is inherited from its parent and so on up the tree.

---

2. Filters

A filter (if it is present) determines whether the **LogRecord** should be forwarded or not. As the name suggests, it filters the log messages according to specific criteria.

A **LogRecord** is only passed from the logger to the log handler and from the log handler to external systems if it passes the specified criteria.

```
// set a filter

logger.setFilter(filter);

// get a filter

Filter filter = logger.getFilter();
```

---

3. Handlers(Appenders)

The log handler or the appenders receive the **LogRecord** and exports it to various targets.

Java SE provides 5 built-in handlers:

Handlers	Use
`StreamHandler`	writes to an `OutputStream`
`ConsoleHandler`	writes to console
`FileHandler`	writes to file
`SocketHandler`	writes to remote TCP ports
`MemoryHandler`	writes to memory

A handler can pass the **LogRecord** to a filter to again determine whether it can be forwarded to external systems or not.

To add a new handler, we use the following code:

```
logger.addHandler(handler);

// example

Handler handler = new ConsoleHandler();

logger.addHandler(handler);
```

To remove a handler, we use the following code:

```
logger.removeHandler(handler);

// example

Handler handler = new ConsoleHandler();

logger.addHandler(handler);

logger.removeHandler(handler);
```

A logger can have multiple handlers. To get all the handlers, we use the following code:

```
Handler[] handlers = logger.getHandlers();
```

---

4. Formatters

A handler can also use a **Formatter** to format the **LogRecord** object into a string before exporting it to external systems.

Java SE has two built-in **Formatters**:

Formatters	Use
`SimpleFormatter`	formats LogRecord to string
`XMLFormatter`	formats LogRecord to XML form

We can use the following code to format a handler:

```
// formats to string form

handler.setFormatter(new SimpleFormatter());

// formats to XML form

handler.setFormatter(new XMLFormatter());
```

LogManager

The **LogManager** object keeps track of the global logging information. It reads and maintains the logging configuration and the logger instances.

The log manager is a singleton, which means that only one instance of it is instantiated.

To obtain the log manager instance, we use the following code:

```
LogManager manager = new LogManager();
```

Advantages of Logging

Here are some of the advantages of logging in Java.

- helps in monitoring the flow of the program

- helps in capturing any errors that may occur
- provides support for problem diagnosis and debugging

Java Assertions

In this tutorial, we will learn about the Java assert statement (Java assertions) with the help of examples.

Assertions in Java help to detect bugs by testing code we assume to be true.

An assertion is made using the `assert` keyword.

Its syntax is:

```
assert condition;
```

Here, `condition` is a boolean expression that we assume to be true when the program executes.

Enabling Assertions

By default, assertions are disabled and ignored at runtime.

To enable assertions, we use:

```
java -ea:arguments
```

OR

```
java -enableassertions:arguments
```

When assertions are enabled and the condition is `true`, the program executes normally.

But if the condition evaluates to `false` while assertions are enabled, JVM throws an `AssertionError`, and the program stops immediately.

Example 1: Java assertion

```
class Main {

  public static void main(String args[]) {

    String[] weekends = {"Friday", "Saturday", "Sunday"};

    assert weekends.length == 2;

    System.out.println("There are " + weekends.length + "  weekends in a week");

  }

}
```

Output

```
There are 3 weekends in a week
```

We get the above output because this program has no compilation errors and by default, assertions are disabled.

After enabling assertions, we get the following output:

```
Exception in thread "main" java.lang.AssertionError
```

Another form of assertion statement

```
assert condition : expression;
```

In this form of assertion statement, an expression is passed to the constructor of the `AssertionError` object. This expression has a value that is displayed as the error's detail message if the condition is `false`.

The detailed message is used to capture and transmit the information of the assertion failure to help in debugging the problem.

Example 2: Java assertion with expression example

```
class Main {

  public static void main(String args[]) {

    String[] weekends = {"Friday", "Saturday", "Sunday"};

    assert weekends.length==2 : "There are only 2 weekends in a week";

    System.out.println("There are " + weekends.length + "  weekends in a week");

  }

}
```

Output

```
Exception in thread "main" java.lang.AssertionError: There are only 2 weekends in a week
```

As we see from the above example, the expression is passed to the constructor of the `AssertionError` object. If our assumption is `false` and assertions are enabled, an exception is thrown with an appropriate message.

This message helps in diagnosing and fixing the error that caused the assertion to fail.

Enabling assertion for specific classes and packages

If we do not provide any arguments to the assertion command-line switches,

```
java -ea
```

This enables assertions in all classes except system classes.

We can also enable assertion for specific classes and packages using arguments. The arguments that can be provided to these command-line switches are:

Enable assertion in class names

To enable assertion for all classes of our program Main,

```
java -ea Main
```

To enable only one class,

```
java -ea:AnimalClass Main
```

This enables assertion in only the `AnimalClass` in the `Main` program.

Enable assertion in package names

To enable assertions for package `com.animal` and its sub-packages only,

```
java -ea:com.animal... Main
```

Enable assertion in unnamed packages

To enable assertion in unnamed packages (when we don't use a package statement) in the current working directory.

```
java -ea:... Main
```

Enable assertion in system classes

To enable assertion in system classes, we use a different command-line switch:

```
java -esa:arguments
```

OR

```
java -enablesystemassertions:arguments
```

The arguments that can be provided to these switches are the same.

Disabling Assertions

To disable assertions, we use:

```
java -da arguments
```

OR

```
java -disableassertions arguments
```

To disable assertion in system classes, we use:

```
java -dsa:arguments
```

OR

```
java -disablesystemassertions:arguments
```

The arguments that can be passed while disabling assertions are the same as while enabling them.

Advantages of Assertion

1. Quick and efficient for detecting and correcting bugs.
2. Assertion checks are done only during development and testing. They are automatically removed in the production code at runtime so that it won't slow the execution of the program.
3. It helps remove boilerplate code and make code more readable.
4. Refactors and optimizes code with increased confidence that it functions correctly.

When to use Assertions

1. Unreachable codes

Unreachable codes are codes that do not execute when we try to run the program. Use assertions to make sure unreachable codes are actually unreachable.

Let's take an example.

```
void unreachableCodeMethod() {

  System.out.println("Reachable code");

  return;
```

```
  // Unreachable code
  System.out.println("Unreachable code");
  assert false;
}
```

Let’s take another example of a switch statement without a default case.

```
switch (dayOfWeek) {
  case "Sunday":
    System.out.println("It’s Sunday!");
    break;
  case "Monday":
    System.out.println("It’s Monday!");
    break;
  case "Tuesday":
    System.out.println("It’s Tuesday!");
    break;
  case "Wednesday":
    System.out.println("It’s Wednesday!");
    break;
  case "Thursday":
    System.out.println("It’s Thursday!");
    break;
  case "Friday":
    System.out.println("It’s Friday!");
    break;
  case "Saturday":
    System.out.println("It’s Saturday!");
    break;
}
```

The above switch statement indicates that the days of the week can be only one of the above 7 values. Having no default case means that the programmer believes that one of these cases will always be executed.

However, there might be some cases that have not yet been considered where the assumption is actually false.

This assumption should be checked using an assertion to make sure that the default switch case is not reached.

```
default:

    assert false: dayofWeek + " is invalid day";
```

If `dayOfWeek` has a value other than the valid days, an `AssertionError` is thrown.

---

2. Documenting assumptions

To document their underlying assumptions, many programmers use comments. Let's take an example.

```
if (i % 2 == 0) {

    ...

} else { // We know (i % 2 == 1)

    ...

}
```

Use assertions instead.

Comments can get out-of-date and out-of-sync as the program grows. However, we will be forced to update the `assert` statements; otherwise, they might fail for valid conditions too.

```
if (i % 2 == 0) {

   ...

} else {

    assert i % 2 == 1 : i;

    ...

}
```

When not to use Assertions

1. Argument checking in public methods

Arguments in public methods may be provided by the user.

So, if an assertion is used to check these arguments, the conditions may fail and result in `AssertionError`.

Instead of using assertions, let it result in the appropriate runtime exceptions and handle these exceptions.

---

2. To evaluate expressions that affect the program operation

Do not call methods or evaluate exceptions that can later affect the program operation in assertion conditions.

Let us take an example of a list `weekdays` which contains the names of all the days in a week.

```
ArrayList<String> weekdays = new ArrayList<>(Arrays.asList("Sunday", "Monday", "Tuesday",
"Wednesday", "Thursday", "Friday", "Saturday" ));


ArrayList<String> weekends= new ArrayList<>(Arrays.asList("Sunday", "Saturday" ));

assert weekdays.removeAll(weekends);
```

Here, we are trying to remove elements `Saturday` and `Sunday` from the ArrayList `weekdays`.

If the assertion is enabled, the program works fine. However, if assertions are disabled, the elements from the list are not removed. This may result in program failure.

Instead, assign the result to a variable and then use that variable for assertion.

```
ArrayList<String> weekdays = new ArrayList<>(Arrays.asList("Sunday", "Monday", "Tuesday",
"Wednesday", "Thursday", "Friday", "Saturday" ));


ArrayList<String> weekends= new ArrayList<>(Arrays.asList("Sunday", "Saturday" ));

boolean weekendsRemoved = weekdays.removeAll(weekends);assert weekendsRemoved;
```

In this way, we can ensure that all the `weekends` are removed from the `weekdays` regardless of the assertion being enabled or disabled. As a result, it does not affect the program operation in the future.
