CORE JAVA

1

INTRODUCTION TO JAVA

Learning Objectives

- Java Buzzwords
- Java Versions
- Byte Code
- Java Core Packages
- Why Learn Java?
- Summary

JAVA BUZZWORDS

- Simple
- Object oriented
- Distributed
- Multithreaded
- Dynamic
- Architecture neutral
- Portable
- High performance
- Robust
- Secure

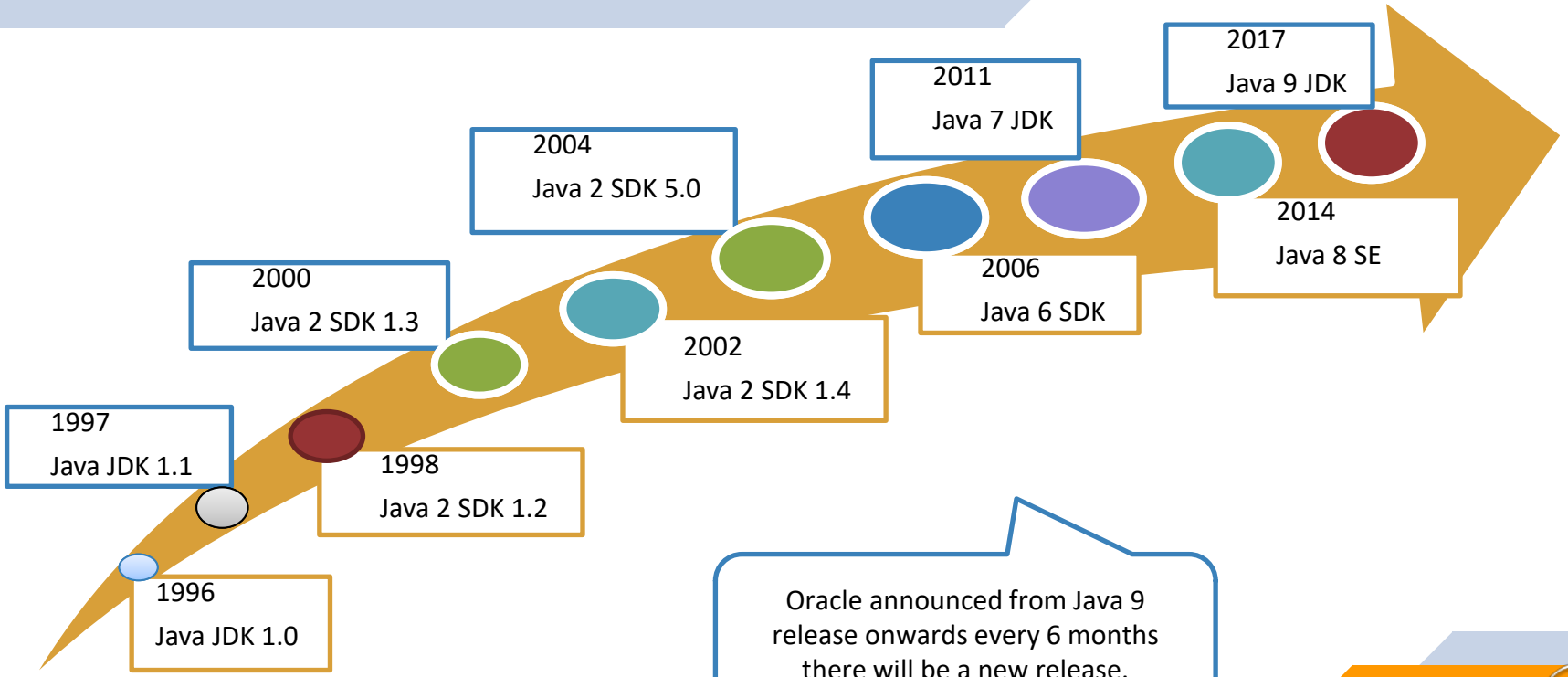
JAVA BUZZWORDS

- Simple
- Object oriented
- Distributed
- Multithreaded
- Dynamic
- Architecture neutral
- Portable
- High performance
- Robust
- Secure

JAVA VERSIONS

- Java JDK 1.0 - 1996
- Java JDK 1.1 - 1997
- Java 2 SDK 1.2 - 1998
- Java 2 SDK 1.3 - 2000
- Java 2 SDK 1.4 - 2002
- Java 2 JDK 5.0 - 2004
- Java 6 JDK – 2006
- Java 7 JDK – 2011
- Java 8 SE – 2014 (LTS)
- Java 9 – Sep 2017

JAVA VERSIONS



Oracle announced from Java 9 release onwards every 6 months there will be a new release.

JAVA VERSIONS

- Java 10 – Mar 2018
- Java 11 – Sep 2018 (LTS)
- Java 12 – Mar 2019
- Java 17 - Sep 2021 (LTS)
- Java 21 – Sep 2023 (LTS)

From Java 11 onwards Oracle created two different licenses, open-source licenses will be available in OpenJDK, and commercial licenses will be available under Oracle.

JAVA VERSION – LTS

- Long-Term Support (LTS):
  - Java 8, Java 11, Java 17, and Java 21 are LTS versions, which receive extended support

From Java 9, Oracle introduced a new release cadence where feature releases are rolled out every six months. Non-LTS versions receive support for six months.

JAVA PROJECTS

Project Loom

- To improve the performance and scalability of Java applications by introducing virtual threads, a lightweight alternative to traditional operating system threads.

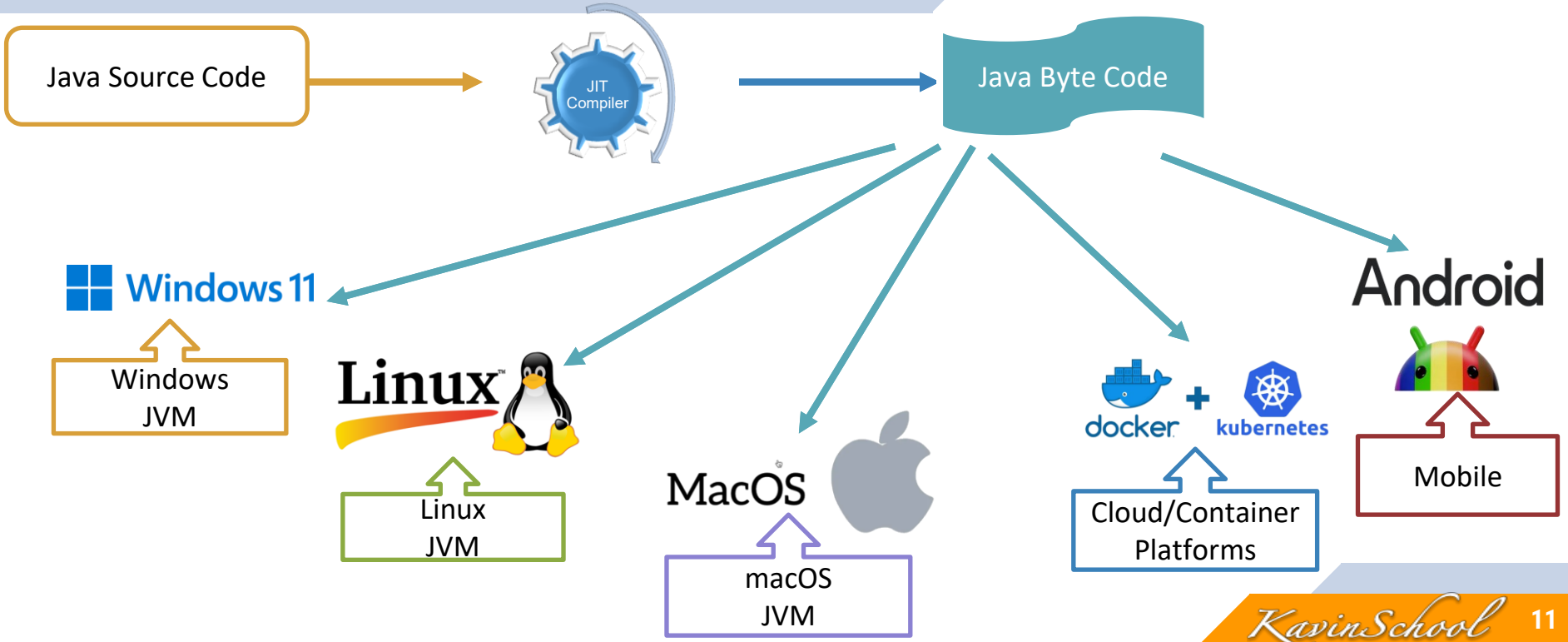
Project Panama

- To bridge the gap between Java and native code, allowing for easier interoperability with C/C++ libraries and APIs.

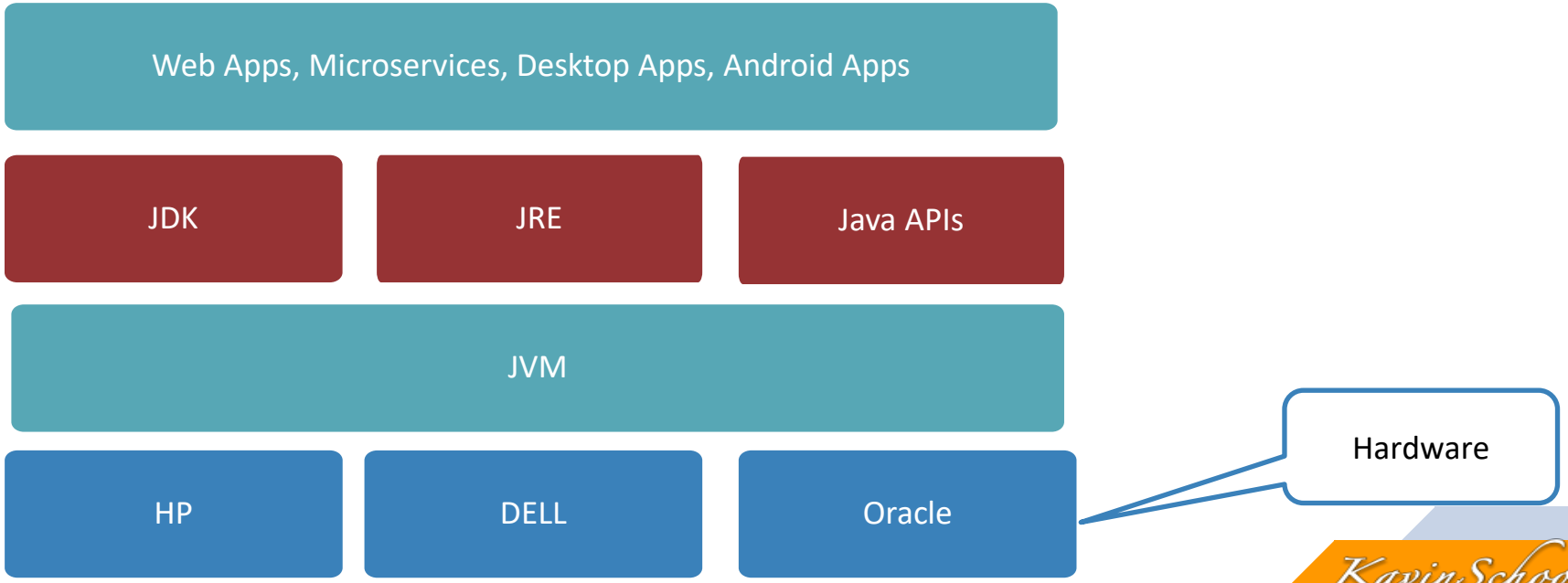
Project Amber

- To introduce language enhancements that improve the expressiveness, readability, and maintainability of Java code.

JAVA BYTE CODE



JAVA PLATFORM



JAVA CORE PACKAGES

java.lang	java.util	java.io	java.time	java.net	java.nio
• The core package contains fundamental classes like Object, String, Math, System, and Thread	• Provides utility classes for collections (List, Set, Map), date and time manipulation, random number generation, and more	• Offers classes for input/output operations, including file handling, streams, and serialization	• Provides a modern and comprehensive API for date and time handling, replacing the older java.util.Date and java.util.Calendar classes	• Contains classes for network programming, such as sockets, URLs, and HTTP connections	• Introduces non-blocking I/O operations for improved performance and scalability

WHY LEARN JAVA?

Widely Used

- Java is used in web development, mobile apps (especially Android), enterprise applications, and much more

Strong Community and Ecosystem

- Thousands of libraries, frameworks, and tools available

Great for Career Development

- Many job opportunities for Java developers

SUMMARY

- Java is an object-oriented, platform-independent, and robust language emphasizing reusability, portability, and maintainability
- The basic structure of a Java program consists of classes and methods
- Java is a versatile language suitable for a wide range of applications

THANKS!

Your feedback is welcome
support@kavinschool.com

2

QE FOCUS ON CODING

Learning Objectives

- The Role of Quality Engineering in Java Development
- Setting Up a Consistent Development Environment
- Adhering to Coding Standards and Conventions
- Code Organization and Readability
- The Impact of Well-Organized Code on Testing

QUALITY ENGINEERING (QE) OVERVIEW

- Java is an object-oriented, platform-independent, and robust language emphasizing reusability, portability, and maintainability.
- The basic structure of a Java program consists of classes and methods.
- Java is a versatile language suitable for a wide range of applications.

QE RESPONSIBILITIES IN JAVA DEVELOPMENT

- Collaborating with developers to ensure quality code
- Designing and writing tests to catch bugs early in the development lifecycle
- Improving code quality through static analysis, peer reviews, and maintaining coding standards

WHY QE IS IMPORTANT?

- Reduces bugs in production
- Improves the maintainability and scalability of the codebase
- Ensures the application behaves as expected under different conditions

SETTING UP A CONSISTENT DEV ENVIRONMENT

- Why Consistency Matters?
  - ✓ A consistent environment ensures that all developers and testers work with the same tools and configurations, reducing discrepancies
  - ✓ Helps avoid "it works on my machine" issues

STEPS TO SET UP A CONSISTENT ENVIRONMENT

JDK Setup

- Ensure all team members use the same Java Development Kit (JDK) version

IDE Configuration

- Agree on a common IDE (e.g., IntelliJ, Eclipse) and its settings like formatting, code styles, and plugins

Build Tools

- Use tools like Maven or Gradle for dependency management and project builds

Dependency Management

- Maintain consistency in libraries and frameworks used

Environment Variables

- Standardize environment variables across development, testing, and production

THE IMPORTANCE OF CODING STANDARDS

- Uniform coding practices improve the readability and maintainability of code
- Enables easier code reviews and testing
- Facilitates onboarding of new developers

JAVA CODING STANDARDS

Naming Conventions

Consistent class, method, and variable names (e.g., camelCase for variables, PascalCase for classes)

Code Formatting

Indentation, bracket placement, and line lengths should follow agreed-upon guidelines

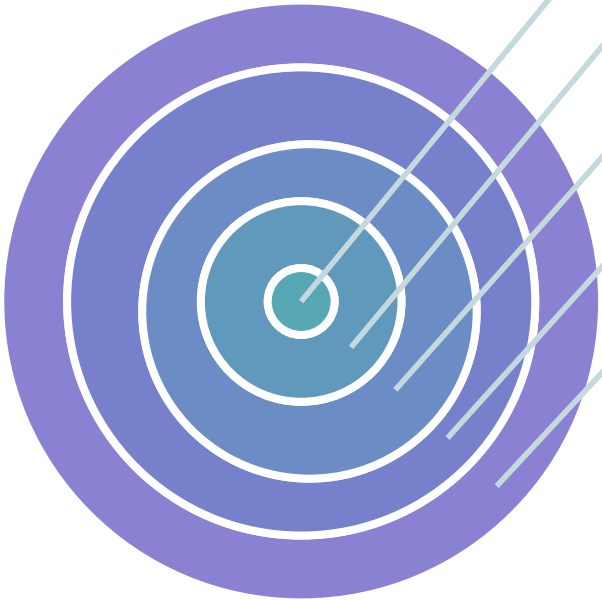
Documentation

Use Javadoc to document methods, classes, and complex logic

CODE ORGANIZATION AND READABILITY

- Why Code Organization Matters?
  - ✓ Well-organized code is easier to read, maintain, and test
  - ✓ Increases collaboration within teams, as others can understand and contribute faster

BEST PRACTICES FOR ORGANIZING JAVA CODE



Package by Feature

- Group related classes by feature rather than by type (e.g., /auth, /payment)

Single Responsibility Principle

- Each class should have one clear purpose

Modularity

- Break large classes and methods into smaller, reusable components

Consistent Naming

- Use meaningful, consistent names for classes, variables, and methods

Avoid Large Methods

- Keep methods short and focused on doing one task

HOW WELL-ORGANIZED CODE ENHANCE TESTING?

Easier Unit Testing

- Well-structured code is easier to isolate and test

Improved Test Coverage

- Code that follows principles like SOLID makes it simpler to write comprehensive tests

Less Duplication

- Organized code reduces redundancy, which simplifies testing and decreases the chance of missed bugs

TESTING APPROACHES IN JAVA

Unit Testing

- Test small pieces of code (JUnit)

Integration Testing

- Test interactions between different modules (TestNG)

End-to-End Testing

- Test the complete flow of an application (Selenium, RestAssured)

VERSION CONTROL IN JAVA PROJECTS

- Why Version Control Matters?
  - ✓ Version control tools like Git allow multiple developers and QEs to collaborate efficiently
  - ✓ Helps track changes, resolve conflicts, and maintain a history of the codebase

BEST PRACTICES FOR VERSION CONTROL

Use Branching Models

- (e.g., GitFlow) to manage features, bug fixes, and releases

Code Reviews

- Use pull requests and code reviews to maintain quality

Commit Messages

- Write clear, concise commit messages to document changes

Continuous Integration

- Integrate frequently to avoid long-running branches and large merge conflicts.

AUTOMATED CODE QUALITY CHECKS

- Why Automate Code Quality Checks?
  - ✓ Automated checks help maintain consistent quality
  - ✓ Automated tools ensure that coding standards and best practices are followed

Manual code reviews can miss issues

JAVA CODE QUALITY TOOLS

Check style

Enforces coding standards

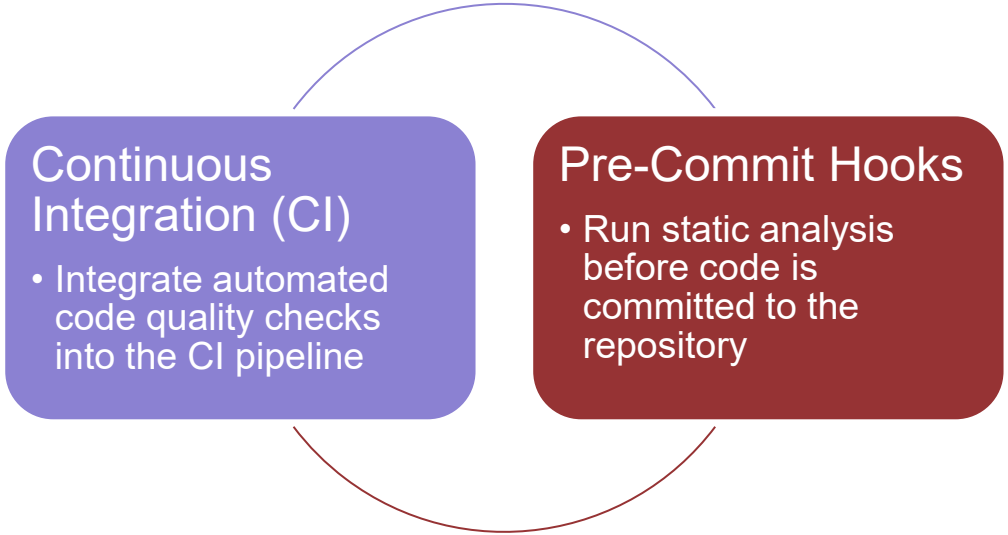
PMD

Detects potential bugs and inefficiencies

SonarQube

Performs static code analysis for quality and security vulnerabilities

AUTOMATING IN CI/CD PIPELINES



SUMMARY

Set Up a Consistent Development Environment	Follow Coding Standards	Organize Code for Readability	Version Control and Collaboration	Automate Quality Checks
Align all team members on tools and configurations	Use best practices for writing clean, readable, and maintainable code	Structure code logically, keeping it modular and easy to test	Use Git and branching strategies for seamless teamwork	Leverage tools like SonarQube, Checkstyle, and automated tests to maintain high-quality standards in Java projects

3

GETTING STARTED WITH JAVA

Learning Objectives

- HelloJava
- Data Types
- Literals
- Operators
- Variables
- Escape Codes

HELLO JAVA PROGRAM

1st
- Type your code
- Check for errors

2nd
- Compile your code
- Use JavaC

3rd
- Run your application
- Use Java

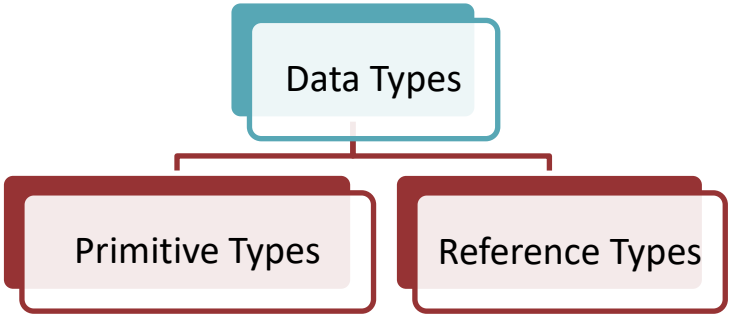
WELCOME

```
package com.kavinschool.corejava.example;

public class Welcome {

public static void main(String[] args) {
   System.out.println("Welcome to Programming");
 }
}
```

The main() method is the starting point

```
Welcome to Programming
```

VARIABLES



- Variables are containers to hold a certain value
- Primitive types are system-defined and are not objects.
  - Ex: int, float, boolean.
- The size and format of the Primitive types are defined by Java and it is system-independent.
- Reference types hold a reference to specific data
  - Ex: Class Types, Interface Types, and Array Types

VARIABLES.JAVA

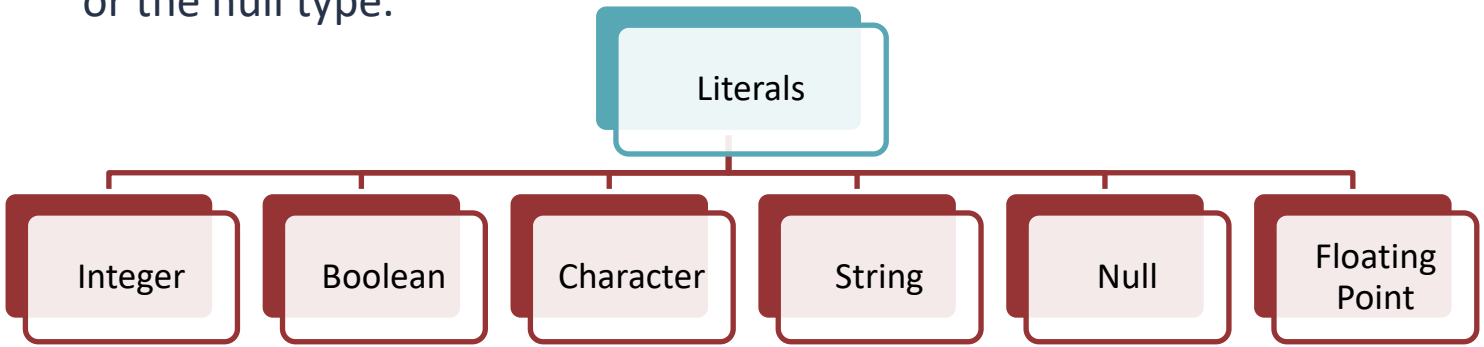
- Find the default value of Primitive types

```
package com.kavinschool.corejava.example;

public class Variables {

static byte k; static int i;static short s; static long l;
static float f; static double d;static char c = 'c'; static boolean b;

public static void main(String args[]) {
  int m = 0;
  System.out.println("byte:" + k + "\nint:" + i + "\nshort:" + s + "\nlong:" + l);
  System.out.println("float:" + f + "\ndouble:" + d + "\nchar:" + c + "\nboolean:" + b);
  System.out.println("m:" + m);
 }
}
```

```
byte:0
int:0
short:0
long:0
float:0.0
double:0.0
char:c
boolean:false
m:0
```

LITERALS

- Literals are used to specify a data value
  - “Hello World” is a string literal
  - ‘c’ is a char literal
  - 1 is an int literal
- A literal is the source code representation of a value of a primitive type, the String type, or the null type.



LITERALS.JAVA

- Example for Literals

```
package com.kavinschool.corejava.example;

public class Literals {

    public static void main(String args[]) {
        byte k = 012;  short s = 0xAB;
        int i = 65000; long l = 65000000L;
        float f = 5.55f;  double d = 6.6666666D;
        char c = 'c';     boolean b = true;

        System.out.println("byte:" + k + "\nint:" + i + "\nshort:" + s + "\nlong:" + l);
        System.out.println("float:" + f + "\ndouble:" + d + "\nchar:" + c + "\nboolean:" + b);
    }
}
```

Pay attention to the L and D at the end of Literals

```
byte:10
int:65000
short:171
long:65000000
float:5.55
double:6.6666666
char:c
boolean:true
```

- Save the code as Literals.Java

ESCAPE CODES

- Escape codes allows you to represent some nongraphic characters and system characters within String and char literals.

Code	Description
\’	Single quotation mark
\”	Double Quotation Mark
\\	Back Slash
\r	Carriage Return
\n	New Line Character
\f	Form Feed
\t	Horizontal Tab
\b	Back Space
\u0000	Unicode Values up to \uFFFF
\000	Octal Values up to \777

OPERATORS

- Operators allows to evaluate mathematical calculations or logical manipulation
- Java has the following Operators

=	>	<	!	~	?	:				
==	<=	>=	!=	&&	||	++	--			
+	-	*	/	&	|	^	%	<<	>>	>>>
+=	-=	*=	/=	&=	|=	^=	%=	<<=	>>=	>>>=

- All binary operators are evaluated from left to right
- Assignment operators are evaluated right to left.

OPERATORS.JAVA

```java
public class Operators {
   public static void main(String args[]) {
      int month = 12;
      int day = 31;
      int total = day * month;
      int exp1 = 1 * 2 * 5 - 3 * 10 - 12 / 2 + 15 % 2;
      double exp2 = 1.0e02 * 2.0e05;
      System.out.println("Total:" + total);
      System.out.println("Exp1:" + exp1);
      System.out.println("Exp2:" + exp2);

      int result = month > day ? 0 : -1;
      System.out.println("result:" + result);
   }
}
```

```
byte:0
int:0
short:0
long:0
float:0.0
double:0.0
char:c
boolean:false
m:0
```

TYPES OF OPERATORS

Arithmetic Operators

- Addition/String Concatenation (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)
- Remainder (%)

Relational Operators

- Equal to (=)
- Not equal to (!=)
- Greater than (>)
- Greater than or equal to (>=)
- Less than (<)
- Less than or equal to (<=)

Type Comparison Operator

- instanceof Compares an object to a specified type

TYPES OF OPERATORS

Logical Operators

- Short-circuit OR (||)
- Short-circuit AND (&&)
- Negation (!)
- Ternary if then else (?:)

Bitwise Operators

- Bitwise AND (&)
- Bitwise OR (|)
- XOR - Bitwise Exclusive OR (^)

Increment/Decrement Operation

- Add a value by 1 (++)
- Subtract a value by 1 (--)

TYPES OF OPERATORS

Bit shift Operators

- Unary bitwise complement (~)
- Signed left shift (<<)
- Signed right shift (>>)
- Unsigned right shift (>>>)

Assignment Operators

- Simple Equal to (=)
- Assignment Operations (+= -= *= /= &= |= ^= %= <<= >>= >>>=)

Other Operators

- Variable Arguments (...)
- Enhanced for loop (:)

THANKS!

Your feedback is welcome
support@kavinschool.com

CONTROL FLOW

4

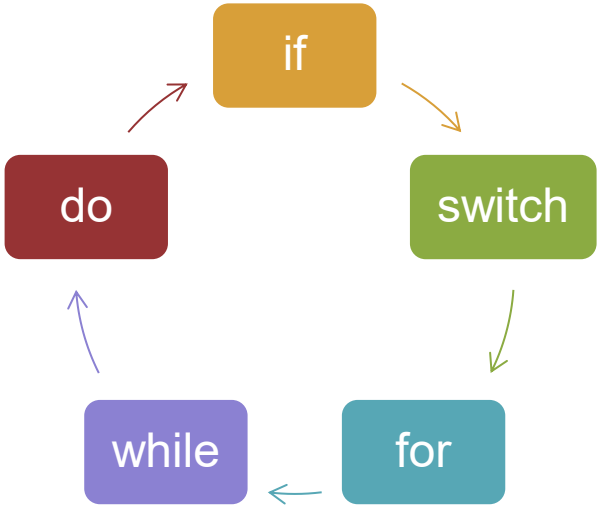
FLOW CONTROL IN JAVA

Learning Objectives

- if-else
- switch

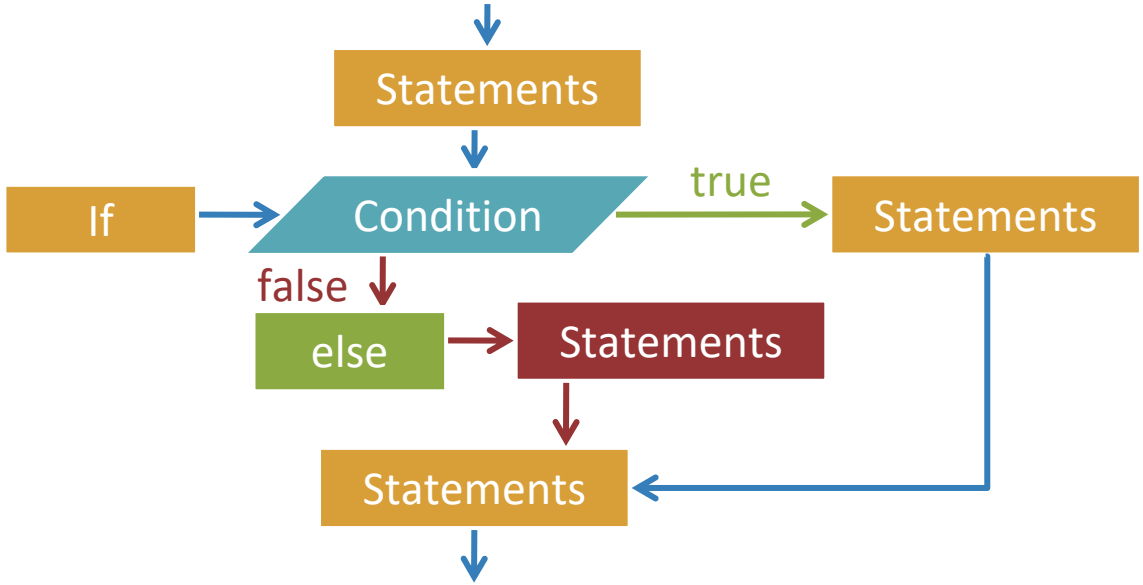
CONTROL FLOW STATEMENTS

- Allows you to execute code by decision making
- Allows you to execute a block in a branch
- Do Loops



IF-ELSE

- Allows you to execute code by decision-making
- Based on a condition certain sections of the statements are executed



IF STATEMENT

- if statements can be written in three ways

Type 1

- if (condition) Statement;

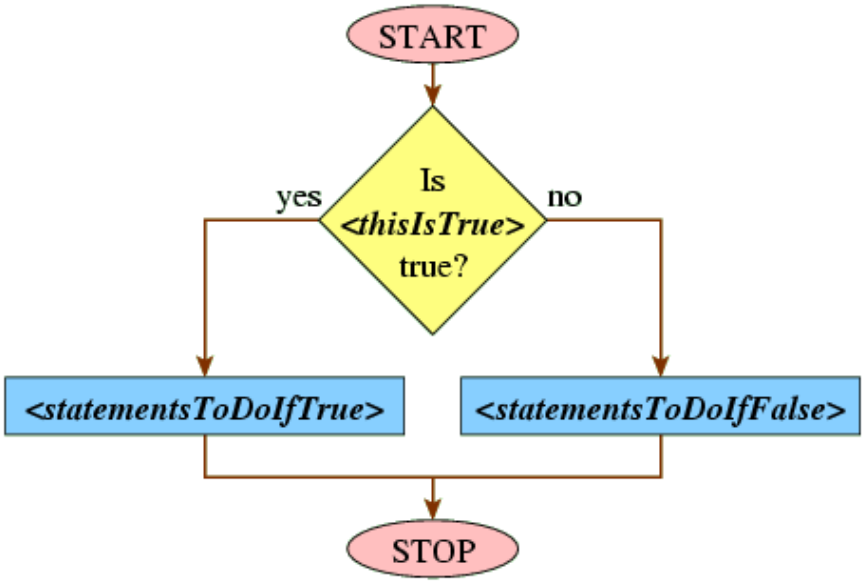
Type 2

- if (condition) {Statements} else {Statements};

Type 3

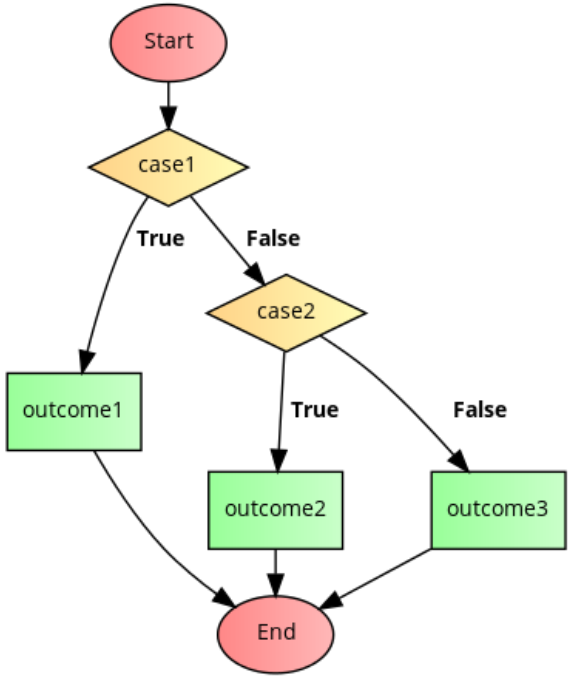
- if (condition) {Statements}
- else if (condition) {Statements}
- else if (condition) {Statements} else {Statements};

IF ELSE


START
Is
<thisIsTrue>
true?
yes
no
<statementsToDoIfTrue>
<statementsToDoIfFalse>
STOP

IF ELSE

```
Start;
if (case1) {outcome1}
else if (case2) {outcome2}
else {outcome3}
End;
```



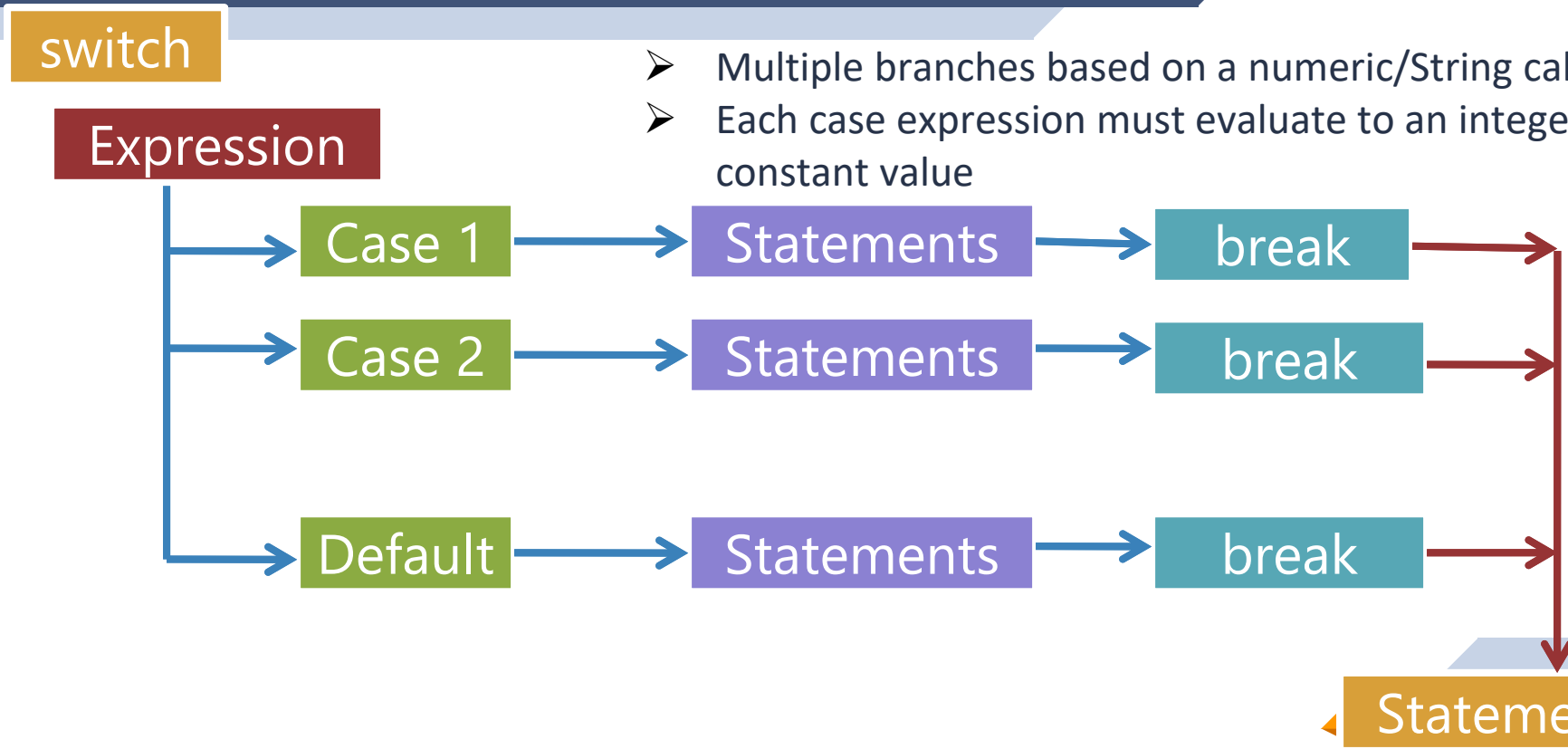
IF STATEMENT

Code

```
public class If {

   public static void main(String args[]) {
      int keith = 5, ken = 15, kavin = 10;
      int max = 0;

      //Simple If
      if (keith > ken) {
         System.out.println("Keith has more money than Ken");
      }

      //Simple If
      if (keith < ken) {
         System.out.println("Ken has more money than Keith");
      }
   }
}
```

Make sure your Class name and file name are always same

SWITCH



- Multiple branches based on a numeric/String calculation
- Each case expression must evaluate to an integer/String constant value

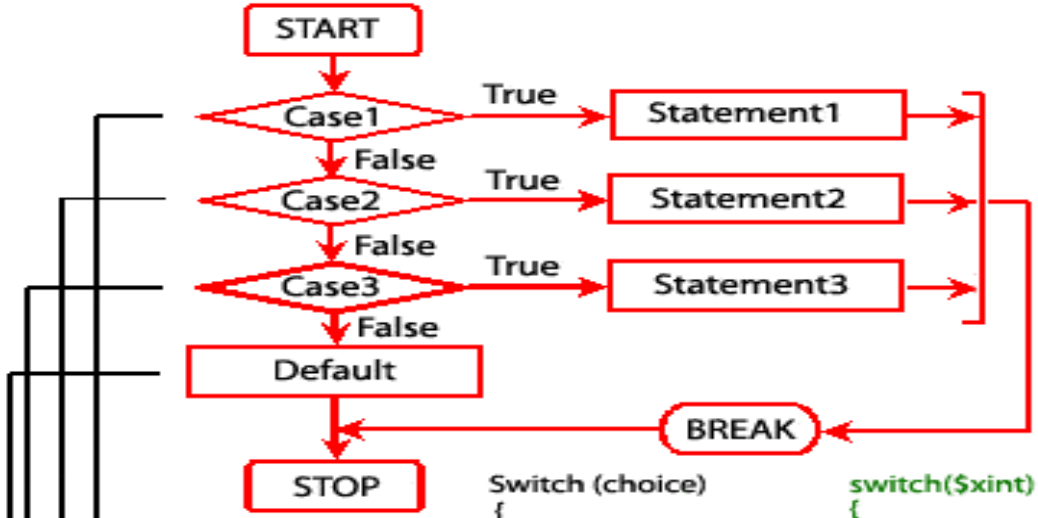
SWITCH

- If no matches found in the case values, executes the (optional) default case if one is provided.
- A switch expression works with byte, short and int primitive data types and enum types and wrapping classes such as Character, Byte, Short and Integer.

Syntax

```
Statements;
switch (expression) {
 case 1: Statements; break;
 case 2: Statements; break;
 default: Statements; break;
}
Statements;
```

Be careful with break; if you forget to add the break statement; the following case statements will be executed automatically




```
Switch (choice)
{
Case constant1:
   Statement1;
   Break;
Case constant2:
   Statement2;
   Break;
Case constant3:
   Statement3;
   Break;
Default:
   Statement Default;
}
```

```
switch($xint)
{
Case 1:
    "This is case No 1.";
    Break;
Case 2:
    "This is case No 2.";
    Break;
Case 3:
    "This is case No 3.";
    Break;
Default:
    "This is default";
}
```

SWITCH.JAVA

- Open NotePad++ and type the below code

Code

```
class Switch {
    public static void main(String[] args) {
        int option = (int) (Math.random() * 6.0);
        System.out.println("Option:"+ option);
        switch (option) {
            case 1:              System.out.println("Selenium IDE"); break;
            case 2:              System.out.println("Selenium Core"); break;
            case 3:              System.out.println("Selenium RC"); break;
            case 4:              System.out.println("Selenium on Rails");
break;
            case 5:              System.out.println("Selenium Grid"); break;
            default:            System.out.println("Selenium"); break;
        }
    }
}
```

Each time provides a random value

```
Option:1
Selenium IDE
```

DIETSCHEDULE.JAVA

- Open NotePad++ and type the below code

Code

```
public class DietSchedule {
   public static void main(String[] args) {
      int option = (int) (Math.random() * 8.0);
      System.out.println("Option:" + option);

      switch (option) {
         case 1: System.out.println("Monday");
         case 2: System.out.println("Tuesday");
         case 4: System.out.println("Thursday");
         case 6: System.out.println("Saturday");
                 System.out.println(" Vegetarian");
                 break;
```

Code

```
case 3: System.out.println("Wednesday");
         case 5: System.out.println("Friday");
         case 7: System.out.println("Sunday");
                 System.out.println(" Non-Vegetarian");
                 break;
         default: System.out.println("Are you on
earth?");
                 break;
      }
   }
}
```

```
Run → 1
Option:0
Are you on earth?

Run → 2
Option:1
Monday
Tuesday
Thursday
Saturday
 Vegetarian
```

5

QE FOCUS ON CONTROL FLOW

Learning Objectives

- Ensuring Code Reliability with Proper Control Structures
- Writing Clear and Concise Methods for Easier Testing and Debugging
- Preventing Bugs Through Effective Use of Java Data Types and Operators

IMPORTANCE OF CONTROL STRUCTURES IN JAVA

- Control structures manage the flow of a program’s execution
- Proper use of control structures ensures that programs behave as expected under different conditions
- Common control structures:
  - Conditionals: **if-else, switch**
  - Loops: **for, while, do-while**
  - Exception Handling**: try-catch-finally**

ENSURING CODE RELIABILITY

- Ensuring Code Reliability with Proper Control Structures
  - ✓ Avoid Nested Conditionals
  - ✓ Switch Expressions (Java 14+)
  - ✓ Loops: Prefer Enhanced For Loop

AVOID NESTED CONDITIONALS

- Deeply nested if-else blocks make code harder to read and maintain
- Solution: Use guard clauses or return early to simplify logic

```
// Before: Nested if
if (isValid(user)) {
    if (hasPermission(user)) {
        performAction(user);
    }
}
```

```
// After: Guard clause
if (!isValid(user)) return;
if (!hasPermission(user)) return;
performAction(user);
```

SWITCH EXPRESSIONS (JAVA 14+)

- Use the new switch expressions to write more concise and exhaustive control flows, reducing the risk of missed cases.

```
int result = switch (status) {
    case SUCCESS -> 1;
    case FAILURE -> 0;
    default -> throw new IllegalStateException("Unknown status");
};
System.out.println("Result: " + result);
```

LOOPS: PREFER ENHANCED FOR LOOP

- Use enhanced for-loops where applicable to reduce off-by-one errors and increase readability.

```
for (String name : namesList) {
    System.out.println(name);
}
```

WHY CONCISE METHODS MATTER?

- Concise methods are easier to understand, test, and debug
- Long methods often hide bugs and make unit testing difficult

BEST PRACTICES FOR WRITING METHODS

- Single Responsibility Principle (SRP)
- Method Length
- Descriptive Method Names
- Return Early to Avoid Deep Nesting

SINGLE RESPONSIBILITY PRINCIPLE (SRP)

- Each method should do one thing and do it well
- Break large methods into smaller ones with clear purposes

```
// Before: A method doing multiple things
public void processOrder(Order order) {
 validate(order); // Responsibility 1: Validation
 calculateDiscount(order); // Resp 2: Business logic for discount
 sendConfirmation(order); // Resp 3: Send confirmation email
}
```

```
// After: Refactor into smaller methods
public void processOrder(Order order) {
    OrderValidator validator = new OrderValidator();
    DiscountCalculator discountCalculator = new DiscountCalculator();
    ConfirmationService confirmationService = new ConfirmationService();

    if (validator.validate(order)) {  // Delegating validation
        discountCalculator.applyDiscount(order);  // Delegating discount calculation
        confirmationService.sendOrderConfirmation(order);  // Delegating confirmation
    }
}
```

SINGLE RESPONSIBILITY PRINCIPLE (SRP)

- Each method should do one thing and do it well
- Break large methods into smaller ones with clear purposes

METHOD LENGTH

- Keep methods under 20-30 lines when possible
- Long methods can be split into helper methods for better clarity and easier testing

DESCRIPTIVE METHOD NAMES

- Use meaningful method names that clearly state their purpose
- Avoid using generic names like **processData()** or **handleInput()**
- Instead, be specific about the action

PREVENTING BUGS JAVA DATA TYPES

- Java Data Types
  - ✓ Primitive Data Types: **int, double, boolean, char**, etc
  - ✓ Reference Data Types: **String, arrays, objects** (classes)
  - ✓ Use the Right Data Type:
    - ❑ Choose data types based on the precision and size needed
    - ❑ Example: Use long instead of int for large numbers

```
long largeNumber = 5000000000L;  // Correct usage for large numbers
float price = 10.5f;  // Correct usage for float
double amount = 100.75;  // Correct usage for double
```

RETURN EARLY TO AVOID DEEP NESTING

- Methods should return as soon as they've completed their task to avoid deep nesting.

```
if (input == null) {
    return;  // Early return for error cases
}
```

COMMON BUGS RELATED TO DATA TYPES

- Integer Overflow
- Floating-Point Precision Errors
- Null Pointer Exceptions

INTEGER OVERFLOW

- Be mindful of data type limits (e.g., **int** has a max value of 2^31 - 1)
- **Solution**: Use data types like long or BigInteger for large values

FLOATING-POINT PRECISION ERRORS

- Use BigDecimal for financial or highly precise calculations instead of double

```
BigDecimal price = new BigDecimal("19.99");
BigDecimal quantity = new BigDecimal("3");
```

NULL POINTER EXCEPTIONS

- Use Optional for nullable values to avoid NullPointerException.

```
Person unknownPerson = null;
String unknownName = Optional.ofNullable(unknownPerson)
.map(Person::name)
.orElse("Unknown Person"); // Provide default if person is null
System.out.println("Unknown Person Name: " + unknownName);
```

OPERATORS AND THEIR CORRECT USE

- Use of Logical Operators
  - ✓ Ensure that boolean logic is correct. For example, don’t confuse & with &&.
- Avoid Using == for Object Comparison
  - ✓ Use .equals() for object comparison instead of ==.

```
if (isUserLoggedIn && isAdmin) {
// Correct usage of short-circuiting
  grantAccess();
}
```

```
String s1 = "Java";
String s2 = "Java";

// Use equals() for object comparison
if (s1.equals(s2)) {
   System.out.println("Strings are equal");
}
```

SUMMARY

- Control Structures
  - Simplify conditionals, use switch expressions, and avoid deep nesting to enhance code clarity and reduce bugs
- Concise Methods
  - Keep methods short and focused on one task to make testing and debugging easier
- Data Types and Operators
  - Prevent bugs by using the correct data types, avoiding null issues, and ensuring proper operator usage

THANKS!

Your feedback is welcome
support@kavinschool.com

6

ANNOTATIONS IN JAVA

Learning Objectives

- Introduction to Java Annotations
- Context Sensitive - JavaDoc

ANNOTATIONS

- Annotations provide data about a program that is not part of the program itself. They have no direct effect on the operation of the code they annotate
- Java has 7 built in annotations
- The java.lang has 3 annotations
  - @Override, @Deprecated, and @SuppressWarnings
- The java.lang.annotation has 4 annotations
  - @Retention, @Documented, @Target, and @Inherited
- Junit, TestNG both uses annotations to identify the test methods

ANNOTATIONS

Annotations	Descriptions
@Inherited	Allows a super-class to be inherited by a sub-class
@Override	Allows a subclass to override a super-class method
@Deprecated	Indicates that a declaration is obsolete and has been replaced by a newer form
@Suppress Warnings	Allows to suppress the compiler warnings. The warnings to suppress are specified by name, in string form

THANKS!

Your feedback is welcome
support@kavinschool.com

7

INTRO TO JAVADOC

Learning Objectives

- JavaDoc

JAVADOC

- Java has three types of comments

A Line comment

```
// This is a line comment
// deep - parameter provides more info
```

A Block Comment

```
/*
*  A multi-line block comment
*/
```

JavaDoc Comment

```
/**
* @author Kangs
*@version 1.0
*/
```

Look the ** in the beginning of multi-line block

JAVADOC TAGS

Tags & Parameters	Usage	Applies to	JDK/SDK
@author *name*	Identifies author of a class	C, I	1.0
{@code text}	In Line HTML are not processed	C,I, F, M	1.5
{@docRoot}	Specifies the path to the root directory of the current documentation.	C,I, F, M	1.3
@deprecated *description*	Specifies that a class or member is deprecated.	M	1.0
@exception *classname desc*	Identifies exception thrown by a method	M	1.0
{@inheritDoc}	Inherits a comment from the immediate super-class.	Overriding M	1.4
{@link *reference*}	Inserts an in-line link to another topic.	C,I, F, M	1.2
{@linkplain *package.class#member label*}	Identical to {@link}, except the link's label is displayed in plain text than code font	C,I, F, M	1.4

Legend

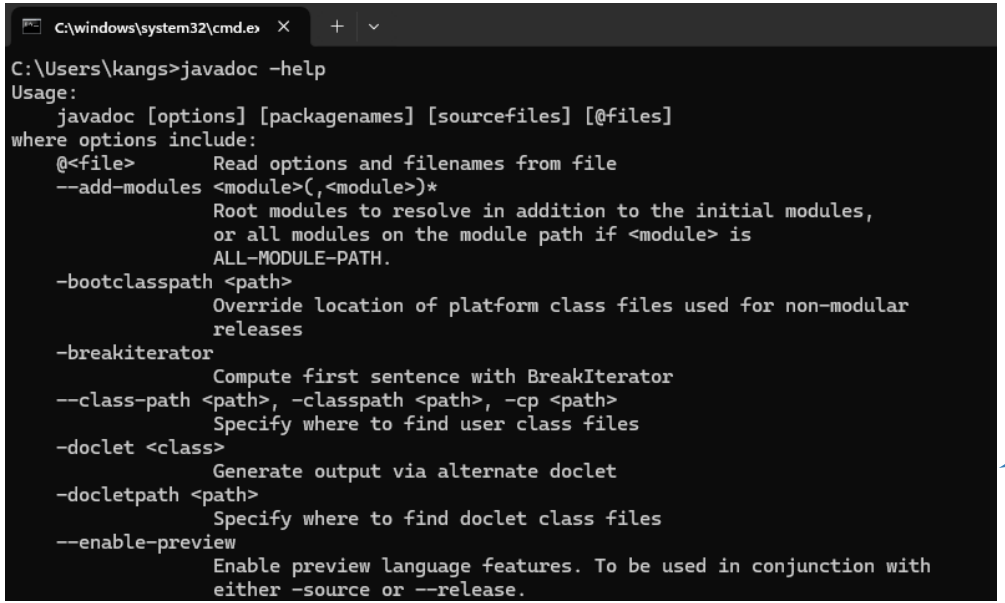
C- Class I-Interface F- Field, M - Method

JAVADOC TAGS

Tags & Parameters	Usage	Applies to	JDK/SDK
{@literal *text*}	Without interpreting HTML markup or nested javadoc tags display text	C,I,F,M	1.5
@param name desc	Method's parameter description	M	1.0
@return desc	Method's return description	M	1.0
@see *reference*	Specifies a link to another topic	C,I,F,M	1.0
@serial *field desc*	Documents a default serializable field.	F	1.2
@serialData *data-description*	Data written by the **writeObject( ) or writeExternal( )** methods information		1.2
@serialField *field name, type, description*	**ObjectStreamField** component Info	F	1.2
@since *text*	Release when introduced	C,I,F,M	1.1
@throws *classN, desc*	Synonym for @exception	M	1.2
{@value *package.class#field*}	Constant Static Field value info	Static Field	1.4
@version *text*	Specifies the version of a class.	C,I	1.0

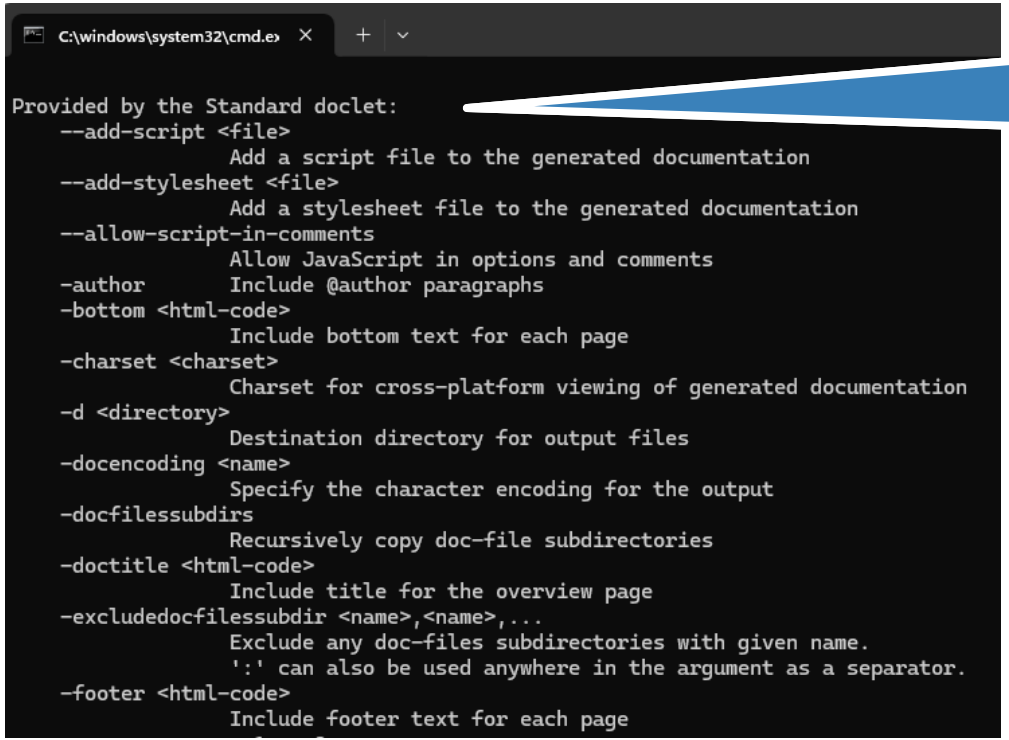
JAVADOC OPTIONS

- Go to Start → Run → cmd
- C:> javadoc –help



All the javadoc options are displayed here

JAVADOC DOCLET OPTIONS



All the javadoc doclet options are displayed here

SHAPE PACKAGE JAVADOC

- In the Shape super class add the below javadoc comments

Code

```
/**
 *
 * @author Kangeyan Passoubady (Kangs) - <a href="http://www.kavinschool.com/">Kavin School </a>
 *
 * @see Shape2D
 * @see Shape3D
 *
 */
```

You can use @see for linking sub-classes.

@author uses HTML <a> tag

SHAPE PACKAGE JAVADOC

- In the Circle class add the below javadoc @deprecated comment, you can @link the replaced method

Code

```
/**
     * Finds the circumference of a Circle
     *
     * @deprecated  Not for public use.
     *    This method is expected to be retained only as
package
     *    private method.  Replaced by
     *    {@link #perimeter}
     */
 @Deprecated
   private double circumference() {
      return 2 * Math.PI * radius;
   }
```

@depricated should be followed by @Depricated in the method

Method circumference is replaced by perimeter, so depricated

```
/**
 * Finds the circumference of a Circle.
 *
 * @deprecated  Not for public use.
 *    This method is expected to be retained only as a package
 *    private method.  Replaced by
 *    {@link #perimeter}
 */
@Deprecated
private double circumference() {
    return 2 * Math.PI * radius;
}
```

Eclipse IDE strike through the deprecated methods.

SHAPE PACKAGE JAVADOC

- In the Square class add the below @return comment
- In the Cuboid class add @param comment

Code

```
/**
     * @return Returns perimeter of a Square
     */
 public double perimeter() {
        return 4 * side;
    }
```

Documents the return of a method

Documents the parameter of a method

Code

```
/**
     * @param deep - accepts boolean value. If true prints
area and perimeter values
     */
    public void draw(boolean deep) {
        draw();
        if (deep) {
            System.out.println("Area: " + area());
            System.out.println("Perimeter: " + perimeter());
        }
    }
```

SHAPE PACKAGE JAVADOC

- Create Package.html if needed

Code

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html lang="en">
   <head>
      <meta http-equiv="content-type" content="text/html; charset=ISO-8859-1"><title>
         Package Overview (Shape)
      </title>
   </head>
   <body>
      <p> Shape package has 3 Abstract classes, namely shape, Shape2D and Shape3D.
<br>
         Shape2D contains Square, Rectangle and Circle <br>
         Shape3D contains Cone, Cube and Cuboid
      </p>
   </body>
</html>
```

If needed document the package information

SHAPE PACKAGE JAVADOC

- Alternatively, you can use Package-info.java if needed

Code

```
/**
 *<p>Shape package has 3 Abstract classes, namely shape, Shape2D and Shape3D.
<br>
 *Shape2D contains Square, Rectangle and Circle <br>
 *Shape3D contains Cone, Cube and Cuboid
 *@author Kangeyan Passoubady (Kangs) - <a
href="http://www.kavinschool.com/">Kavin School </a>
 *@see Shape
 *</p>
 *
 */
package shape;
```

If needed document the package information

SHAPE OVERVIEW JAVADOC

- Create overview.html if needed

Code

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html lang="en"><head>
        <meta http-equiv="content-type" content="text/html; charset=ISO-8859-1"><title>
            Overview (Shape)
        </title>
    </head>
    <body>
        <p>Shape is an example project helps to understand the inheritance and
polymorphism of Java OOPS concepts</p>
        <img src="Shape Class Hierarchy.png" width="1006" height="823" alt="Shape Class
Hierarchy"/>
    </body>
</html>
```

If needed document the package information

SHAPE OVERVIEW JAVADOC

- Provide @version and @since in ShapeDemo

Code

```
/**
 *
 * @author Kangeyan Passoubady (Kangs)
 * - http://www.kavinschool.com
 * @see shape.Cone
 * @see shape.Cube
 * @see shape.Circle
 * @see shape.Cuboid
 * @see shape.Rectancle
 * @see shape.Square
 * @since 1.0
 * @version 1.0
 */
public class ShapeDemo {
```

Provide @see links to different classes

SHAPE OVERVIEW JAVADOC

- Use @serial to document the attributes
- Use @value to document the constants

Code

```
/**
     * @serial holds the total number of shapes object created. Note this will
contain total of both Shape2D and Shape3D objects.
     */
    static int totalShapes = 0;
    /**
     * @value NAME holds "Shape" string
     */
    static final String NAME = "Shape";
```

@value is used for static variables

CONTEXT SENSITIVE JAVADOC

```
@Override
public void draw() {
    super.draw();
    System.out.println("Name:" + NAME);
    System.out.println("Type: " + type);
    System.out.println("Height: " + height);
    System.out.println("Width: " + width);
    System.out.println("Length: " + length);
}
}
```

Add @override when you override super class methods

```
 */
@Deprecated
private double circumference() {
    return 2 * Math.PI * radius;
}
```

Add @Deprecated makes the method deprecated

```
@SuppressWarnings("deprecation")
private double circumference() {
    return 2 * length + 2 * breadth;
}
```

Add @SuppressWarnings (“deprecation”) to hide the compiler warnings

ANNOTATIONS

Annotations	Descriptions
@Inherited	Allows a super-class to be inherited by a sub-class
@Override	Allows a subclass to override a super-class method
@Deprecated	Indicates that a declaration is obsolete and has been replaced by a newer form
@Suppress Warnings	Allows to suppress the compiler warnings. The warnings to suppress are specified by name, in string form

8

LESSON

LOOPS

LOOP

LOOP STATEMENTS

A loop is a set of commands that executes repeatedly until a specified condition is met.

- for statement
- for each statement
- while statement
- do .. while statement

Loop Statements

- break statement
- continue statement

Loop Assisting Statements

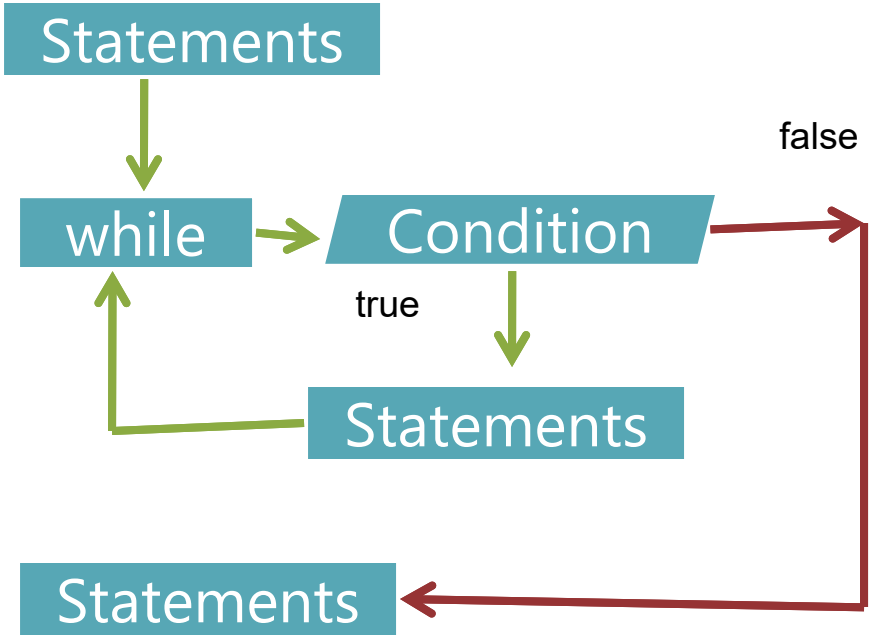
WHILE

- While loop
  - ✓ when a condition is true executes a bunch of statements
- while loop
  - ✓ will not execute the statement even for the first time when the condition evaluated becomes false

```
System.out.println("Welcome to Loops!!!");
int i = 0;
while (i < 100) {
    if (i == 10) {
        break; // terminate loop if i is 10
    }
    System.out.println("i: " + i);
    i++;
}
System.out.println("Loop complete.");
```

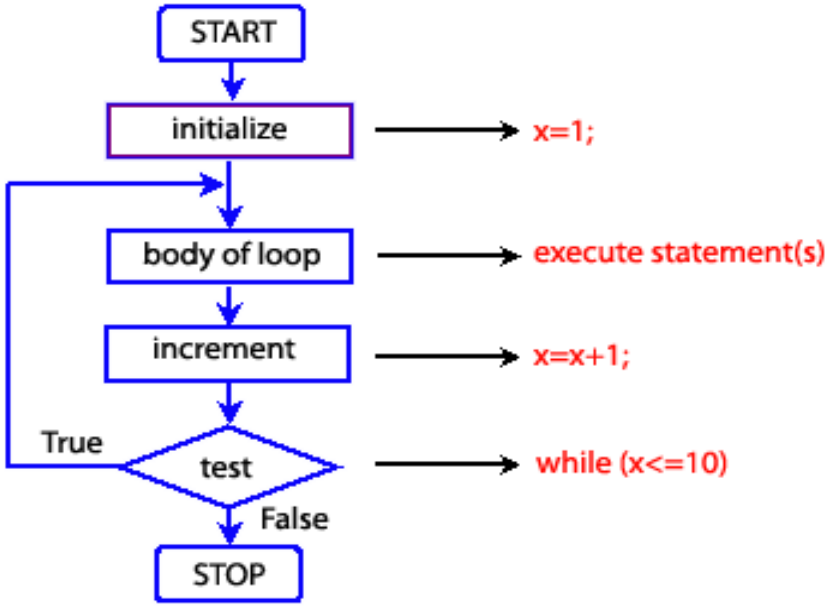
```
Welcome to Loops!!!
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
Loop complete.
```

WHILE



While loop statements are executed until the condition becomes false

WHILE LOOP


START
initialize
x=1;
body of loop
execute statement(s)
increment
x=x+1;
True
test
while (x<=10)
False
STOP

WHILE

- Use a **break** to come out of the while loop
- Use **continue** to skip the current iteration
- Avoid infinite loops by exercising appropriate end condition

Syntax

```
Statements;
while (condition) {
  Statements;
}
Statements;
```

If your condition never becomes **false,** then you may inadvertently create an infinite loop

DO..WHILE

- do-while loop checks the condition at the bottom of the loop
- do-while statements are guaranteed to execute at least once

```
int count = 0;
System.out.println("Before Loop Starts.");
do {
    //if (count%2==0) continue;
    System.out.println("Current Value: " +
count);
    if (count == 5) break;
} while (++count <= 10);
System.out.println("After Loop complete.");
```

```
Before Loop Starts.
Current Value: 0
Current Value: 1
Current Value: 2
Current Value: 3
Current Value: 4
Current Value: 5
After Loop complete.
```

LOOP STATEMENTS (DO .. WHILE STATEMENT)

The do...while statement repeats until a specified condition evaluates to **false**

Syntax:
do {statement} while (condition)

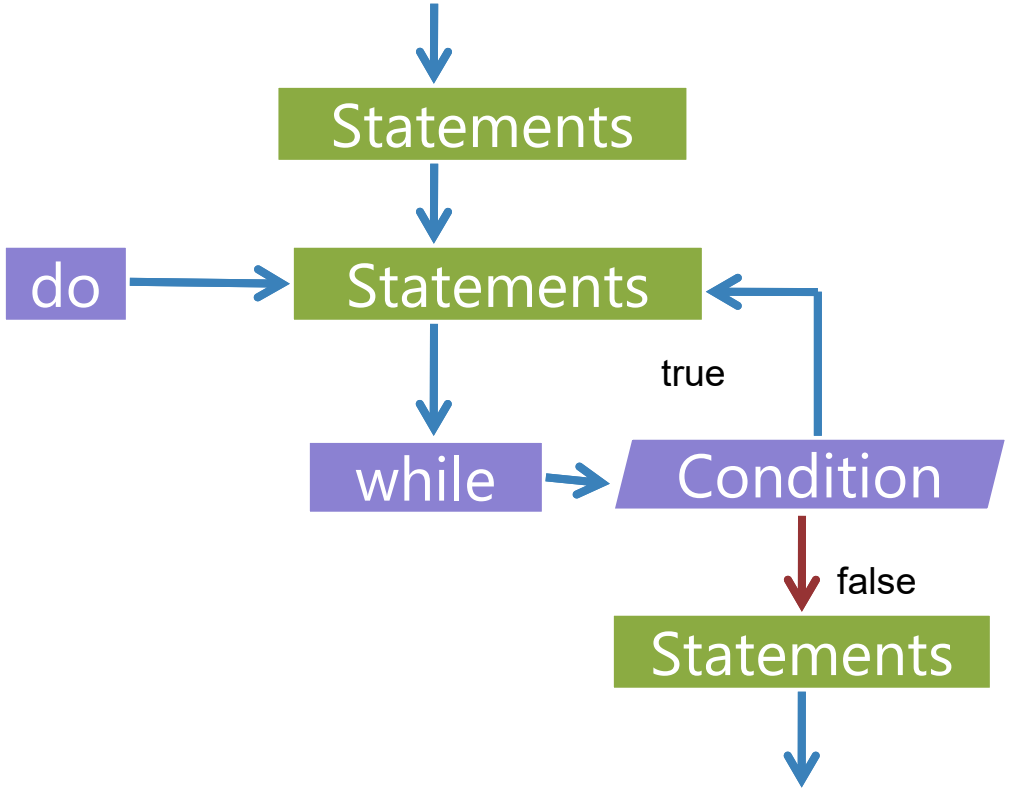
- When a do… while the loop executes, the following occurs:

1. statement executes once before the condition is checked

2. To execute multiple statements, use a block statement ({ ... }) to group those statements

3. If the **condition** is true, the statement executes again. At the end of every execution, the condition is checked
When the condition is false, execution stops, and control passes to the statement following do...while

DO-WHILE



do-while loop statements are executed until the condition becomes true

DO-WHILE

- Use a **break** to come out of the while loop
- Use **continue** to skip the current iteration
- Avoid infinite loops by exercising appropriate end condition

Syntax

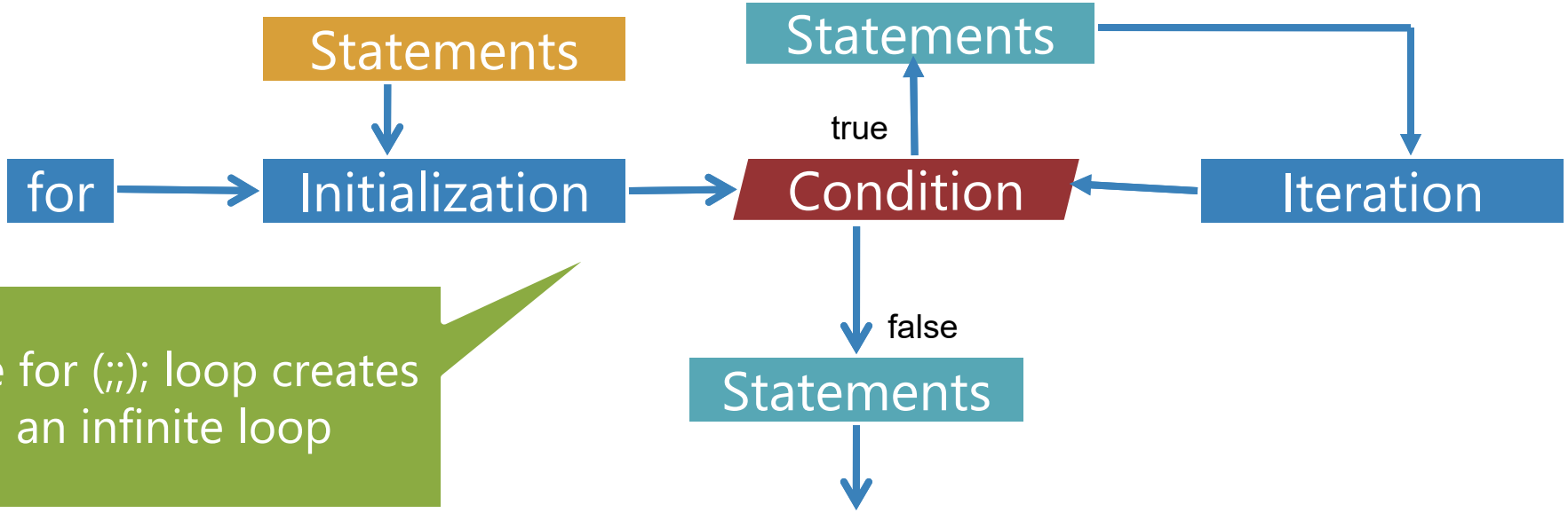
```
Statements;
 do {
  Statements;
} while (condition)
Statements;
```

If your condition never become **false**, then you may inadvertently create an infinite loop

FOR LOOP

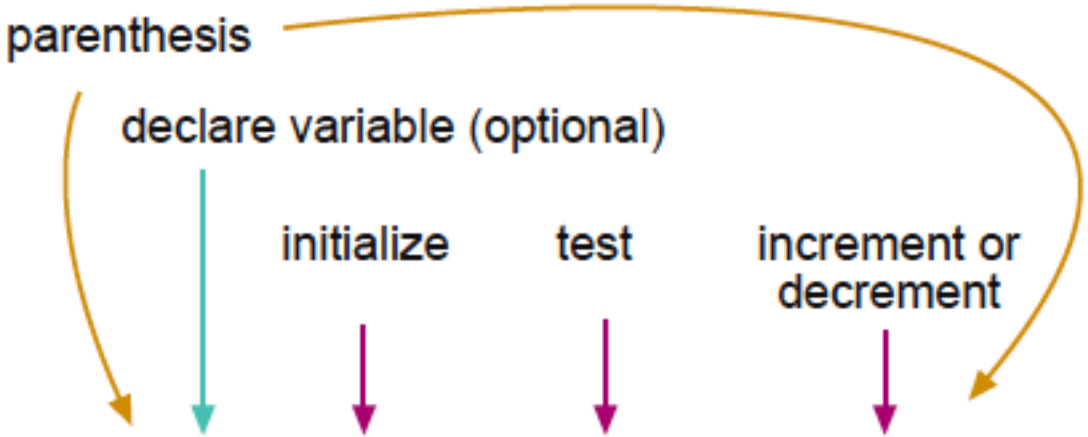
- For loop provides compact way to iterate a ranges of values
- For repeating block contains three sections

1. Initialization
2. Condition
3. Iteration



The for (;;); loop creates an infinite loop

FOR LOOP




```
for(int x = 0; x < 100; x++){

    println(x);  // prints 0 to 99
}
```

FOR LOOP

- **Initialization** → Sets the initial value of the loop control variable(s)
- **Condition** → Boolean expression determines whether the loop should repeat or not
- **Iteration** → Expression(s) which changes loop control variable(s)

Syntax

```
Statements;

for (initialization; condition; iteration) {
  Statements;
}
Statements;
```

Initialization, Condition, and Iteration all are optional

FOR LOOP EXAMPLE

```
public class AsciiChart {
    public static void main(String[] args) {
        System.out.println("============================");
        System.out.println("       ASCII Chart");
        System.out.println("============================");
        System.out.println("Dec\tHex\tOct\tAscii");
        System.out.println("============================");
        for (int count = 0; count <= 255; count++) {
            System.out.printf("%d\t%X\t%o\t%c\n", count, count, count, count);
        }
        System.out.println("============================");
    }
}
```

```
============================
       ASCII Chart
============================
Dec     Hex     Oct     Ascii
============================
0       0       0
1       1       1       ☺
2       2       2       ☻
3       3       3       ♥
4       4       4       ♦
5       5       5       ♣
6       6       6       ♠
7       7       7
8       8       10
9       9       11
10      A       12

11      B       13

12      C       14

13      D       15
14      E       16
15      F       17
16      10      20      ►
17      11      21      ◄
18      12      22      ↕
19      13      23      ‼
20      14      24      ¶
21      15      25      §
22      16      26      ▬
23      17      27      ↨
```

LOOP ASSISTING STATEMENTS (BREAK STATEMENT)

Use the break statement to terminate a **loop**, **switch, or label** statement.

Syntax:

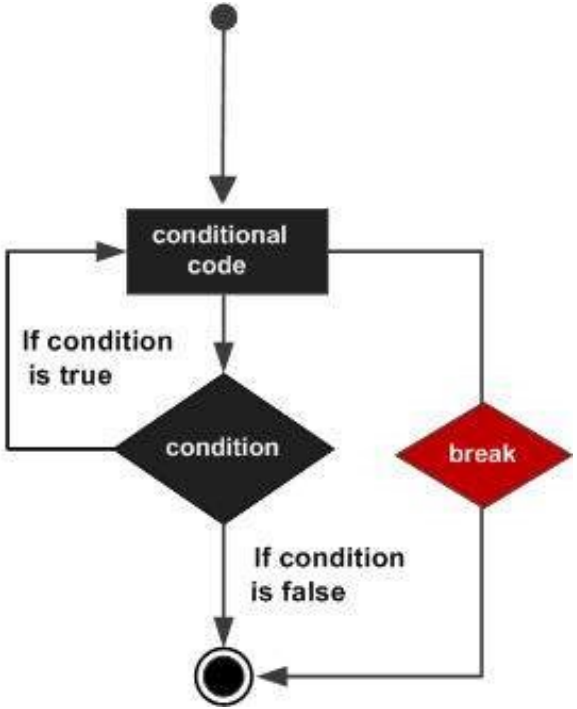
```
break;
break label;
```

```
labelName:
while (condition) {
    // loop body
    if (someCondition) {
        break labelName; // exits the loop labeled by 'labelName'
    }
}
```

When you use a break, it terminates the innermost enclosing **while, do-while, for**, or **switch** immediately and transfers control to the following statement.

When you use **break** with a **label**, it terminates the specified labeled statement.

BREAK


conditional code
If condition is true
condition
break
If condition is false


break

THANKS!

Your feedback is welcome
support@kavinschool.com