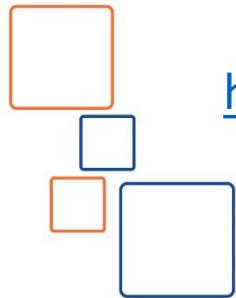


CORE JAVA PROGRAMMING

AN INTRODUCTION TO JAVA



Course link on maharatech:

<https://maharatech.gov.eg/course/view.php?id=2052>



Java™ Education
and Technology Services

Invest In Yourself,
Develop Your Career

A.	The History and Evolution of Java	2
B.	An Overview of Java	21
C.	Data Types, Variables and Arrays	64
D.	Operators, Control Statements and String Handling	98
E.	Packages and Interfaces	163
F.	Wrapper Classes, Enumeration, Autoboxing and Annotations	216
G.	Exception Handling	256
H.	Generics and Lambda Expressions	299
I.	Java Stream API	363
J.	Java Collections	430
K.	Java Platform Module System	522

Lesson 1

An Overview of Java's History and Evolution

The History and Evolution of Java

HISTORY OF JAVA

James Gosling, Mike Sheridan, and Patrick Naughton initiated the Java language project in June 1991. The small team of sun engineers called **Green Team**.



James Gosling



The History and Evolution of Java

- An easier—and more cost-efficient—solution was needed.
- Gosling and others began work on a **portable, platform-independent** language that could be used to produce code that would run on a variety of CPUs under differing environments.
- This effort ultimately led to the creation of **Java**.
- A second, and ultimately more important, factor was emerging that would play a crucial role in the future of Java. This second force was, of course, the World Wide Web.
- However, with the emergence of the World Wide Web, Java was pushed to the forefront of computer language design, because the Web, too, demanded portable programs.
- **Portable (platform-independent) programs is nearly as old as the discipline of programming itself**, it had taken a back seat to other, more pressing problems. Further, because (at that time) much of the computer world had divided itself into the three competing camps of **Intel, Macintosh, and UNIX**, most programmers stayed within their fortified boundaries, and the urgent need for portable code was reduced.
- By 1993, it became obvious to members of the Java design team that the problems of portability frequently encountered when creating code for embedded controllers are also found when attempting to create code for the Internet.
- **This realization caused the focus of Java to switch from consumer electronics to Internet programming**. So, while the desire for an **architecture-neutral** programming language provided the initial spark, the Internet ultimately led to Java's large-scale success.

The History and Evolution of Java

- Java derives much of its character from C and C++. This is by intent. The Java designers knew that using the familiar syntax of C and echoing the object-oriented features of C++ would make their language appealing to the legions of experienced C/C++ programmers.
- Java shares some of the other attributes that helped make C and C++ successful.
 - Java was designed, tested, and refined by real, working programmers. It is a language grounded in the needs and experiences of the people who devised it. Thus, Java is a programmer's language.
 - Java is cohesive and logically consistent.
 - except for those constraints imposed by the Internet environment, Java gives you, the programmer, full control. **If you program well, your programs reflect it. If you program poorly, your programs reflect that, too.**

Java is not a language with training wheels. It is a language for professional programmers.

- The environmental change that prompted Java was the need for platform-independent programs designed for distribution on the Internet.
- Java also embodies changes in the way that people approach the writing of programs. For example, Java enhanced and refined the object-oriented paradigm used by C++, added integrated support for multithreading, and provided a library that simplified Internet access.

The History and Evolution of Java

Java was the perfect response to the demands of the then newly emerging, highly distributed computing universe.

Java was to Internet programming what C was to system programming:

A revolutionary force that changed the world.

The History and Evolution of Java

How Java Impacted the Internet

- The Internet helped pushing Java to the forefront of programming, and Java, in turn, had a deep effect on the Internet.
- In addition to simplifying web programming in general, Java innovated a new type of networked program called the *applet* that changed the way the online world thought about content. Java also addressed some of the trickiest issues associated with the Internet: portability and security. Let's give some brief information about each of these:
 - **Java Applets**
 - **Security**
 - **Portability**

The History and Evolution of Java

How Java Impacted the Internet

- **Java Applets**
 - At the time of Java's creation, one of its most exciting features was the applet. An *applet* is a special kind of Java program that is designed to be transmitted over the Internet and automatically executed inside a Java-compatible web browser.
 - They were typically used to display data provided by the server, handle user input, or provide simple functions, such as a loan calculator, that execute locally, rather than on the server. In essence, the applet allowed some functionality to be moved from the server to the client.
 - In the early days of Java, applets were a crucial part of Java programming.
 - **Starting with JDK 9, applets are being phased out and deprecated**, with other mechanisms supplying an alternative way of delivering dynamic, active programs via the Web.
- **Security**
- **Portability**

The History and Evolution of Java

How Java Impacted the Internet

- **Security**
 - As you are likely aware, every time you download a “normal” program, you are taking a risk, because the code you are downloading might contain a virus, Trojan horse, or other harmful code.
 - Java achieved this protection by enabling you to confine an application to the Java execution environment and prevent it from accessing other parts of the computer.
 - The ability to download programs with a degree of confidence that no harm will be done may have been one of the the most innovative aspect of Java at that time.

The History and Evolution of Java

How Java Impacted the Internet

- **Portability**
 - Portability is a major aspect of the Internet because there are many different types of computers and operating systems connected to it.
 - If a Java program were to be run on virtually any computer connected to the Internet, there needed to be some way to enable that program to execute on different systems.
 - The same code must work on all computers. Therefore, some means of generating portable executable code was needed.
 - The same mechanism that helps ensure security also helps create portability.

Java's Magic: The Bytecode

- The key that allows Java to solve both the security and the portability problems just described is that **the output of a Java compiler is not executable code. Rather, it is bytecode.**
- *Bytecode* is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the **Java Virtual Machine (JVM)**. In essence, the original JVM was designed as an interpreter for bytecode.
- However, the fact that a Java program is executed by the JVM helps solve the major problems associated with web-based programs.

Java's Magic: The Bytecode

- Translating a Java program into bytecode makes it much easier to run a program in a wide variety of environments because only the JVM needs to be implemented for each platform.
- Once the run-time package exists for a given system, any Java program can run on it. Remember, although the details of the JVM will differ from platform to platform, all understand the same Java bytecode.
- The fact that a Java program is executed by the JVM also helps to make it secure. Because the JVM is in control, it manages program execution.

Java's Magic: The Bytecode

- It is possible for the JVM to create a restricted execution environment, called the **sandbox**, that contains the program, preventing unrestricted access to the machine.
- Although Java was designed as an interpreted language, there is nothing about Java that prevents on-the-fly compilation of bytecode into native code in order to boost performance.

The Java Buzzwords

- No discussion of Java's history is complete without a look at the Java buzzwords.
- The key considerations were summed up by the Java team in the following list of buzzwords:

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

The Java Buzzwords

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

- **Simple**

- Java was designed to be easy for the professional programmer to learn and use effectively.
- If you already understand the basic concepts of object oriented programming, learning Java will be even easier.
- if you are an experienced C++ programmer, moving to Java will require very little effort.

- **Object-Oriented**

- The object model in Java is simple and easy to extend, while primitive types, such as integers, are kept as high-performance non objects.

The Java Buzzwords

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

- **Robust**

- Because Java is a strictly typed language, it checks your code at compile time. However, it also checks your code at run time.
 - To better understand how Java is robust, consider two of the main reasons for program failure: memory management mistakes and mishandled exceptional conditions (that is, run-time errors).

- **Multithreaded**

- Java supports multithreaded programming, which allows you to write programs that do many things simultaneously.
- The Java run-time system comes with an elegant yet sophisticated solution for multiprocess synchronization that enables you to construct smoothly running interactive systems.

The Java Buzzwords

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

- **Architecture-neutral**
 - Their goal was “*write once; run anywhere.*” To a great extent, this goal was accomplished.
- **Interpreted and High Performance**
 - Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java **bytecode**.
 - This code can be executed on any system that implements the Java Virtual Machine.
 - The Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler.

The Java Buzzwords

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

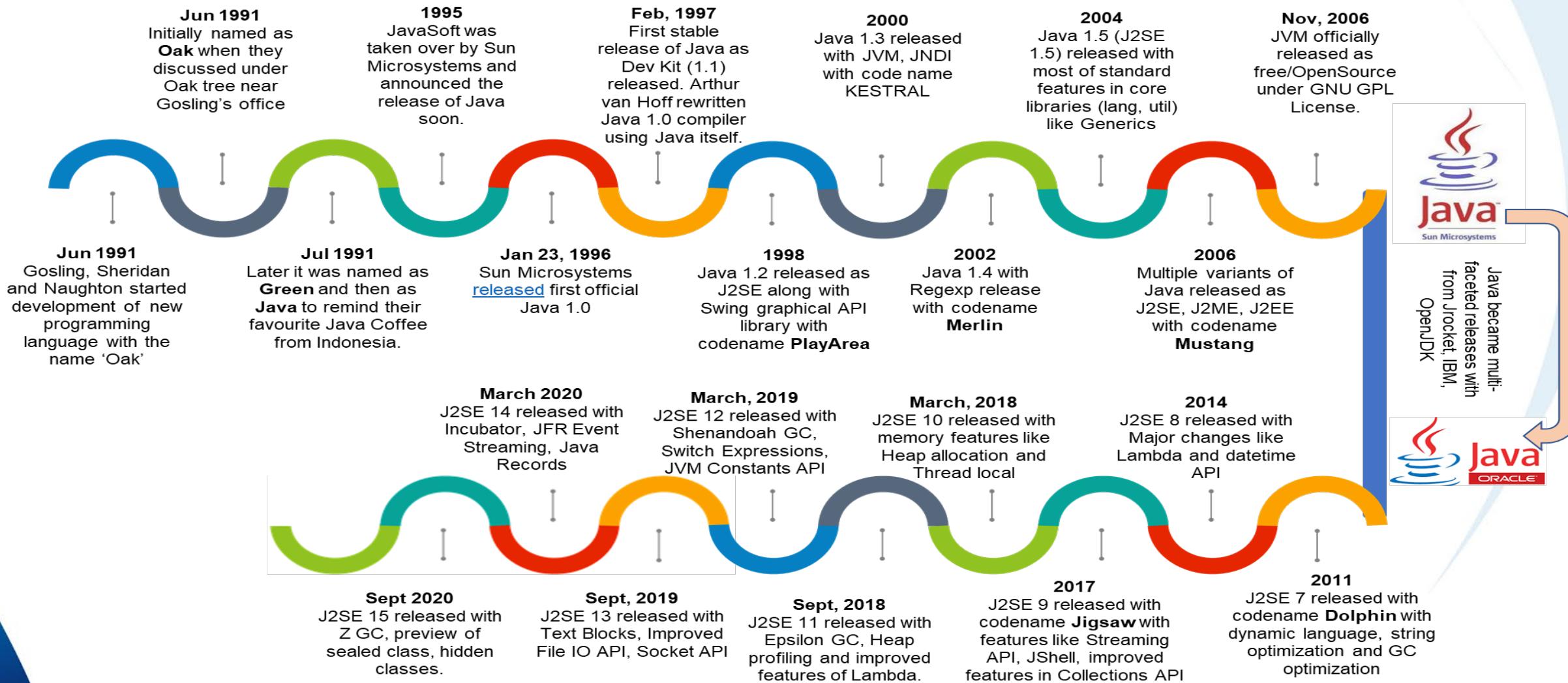
- **Distributed**

- Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols.
- Java also supports Remote Method Invocation (RMI), this feature enables a program to invoke methods across a network.

- **Dynamic**

- Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve access to objects at run time. This makes it possible to **dynamically link** code in a safe and convenient manner. This is crucial to the robustness of the Java environment, in which small fragments of bytecode may be dynamically updated on a running system.

The Evolution of Java



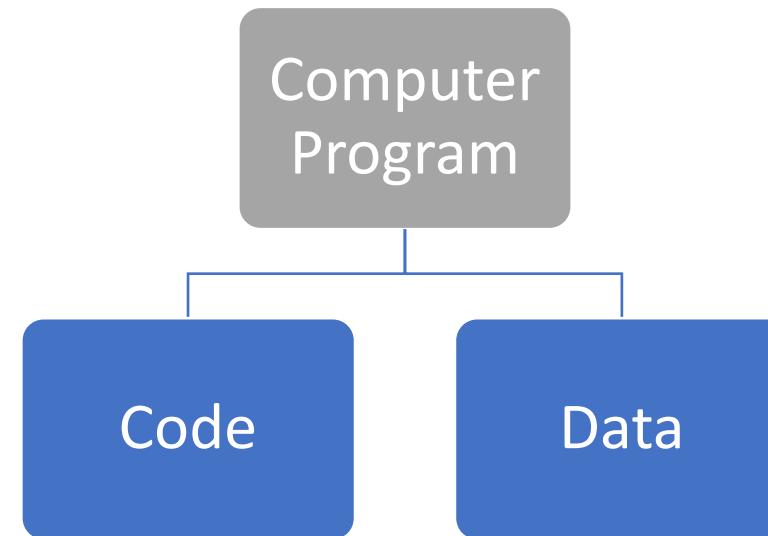
A.	The History and Evolution of Java	2
B.	An Overview of Java	21
C.	Data Types, Variables and Arrays	64
D.	Operators, Control Statements and String Handling	98
E.	Packages and Interfaces	163
F.	Wrapper Classes, Enumeration, Autoboxing and Annotations	216
G.	Exception Handling	256
H.	Generics and Lambda Expressions	299
I.	Java Stream API	363
J.	Java Collections	430
K.	Java Platform Module System	522

An Overview of Java

- Object-Oriented Programming
 - Object-oriented programming (OOP) is at the core of Java.
 - OOP is so integral to Java that it is best to understand its basic principles before you begin writing even simple Java programs.

Computer Program Components

- These are the two paradigms that govern how a program is constructed.



- The first way is called the process-oriented model.
- The second approach, called object-oriented programming

The Process-Oriented model

- The process oriented model can be thought of as code acting on data.
- Procedural languages such as C employ this model to considerable success.
- However, the problem with this approach appear as programs grow larger and more complex.

The Object-Oriented Programming

- Object-oriented programming organizes a program around its data (that is, objects) and a set of well-defined interfaces to that data.
- An object-oriented program can be characterized as data controlling access to code.
 - *We will be discussing the main characteristics of Object Oriented*

Abstraction

- An essential element of object-oriented programming is abstraction.
- Humans manage complexity through abstraction.
- For example, people do not think of a **car** as a set of tens of thousands of individual parts. They think of it as a well-defined object with its own unique behavior.
- A powerful way to manage abstraction is through the use of hierarchical classifications.
- This allows you to layer the semantics of complex systems, breaking them into more manageable pieces.

Abstraction

- From the outside, the car is a single object.
- Once inside, you see that the car consists of several subsystems: **steering, brakes, sound system, seat belts, heating, cellular phone**, and so on. In turn, each of these subsystems is made up of more specialized units.
- Hierarchical abstractions of complex systems can also be applied to computer programs.
- The data from a traditional process-oriented program can be transformed by abstraction into its component objects.
- A sequence of process steps can become a collection of messages between these objects.

Abstraction

- Thus, each of these objects describes its own unique behavior.
- You can treat these objects as concrete entities that respond to messages telling them to *do something*.
- As you will see, object-oriented programming is a powerful and natural paradigm for creating programs that survive the changes accompanying the life cycle of any major software project, including conception, growth, and aging.

The Three OOP Principles

- All object-oriented programming languages provide mechanisms that help you implement the object-oriented model.
- They are ***encapsulation***, ***inheritance***, and ***polymorphism***.
- In the following slides we will be discussing these concepts briefly.

Encapsulation

- Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse.
- In Java, the basis of encapsulation is the class.
- A class defines the structure and behavior (data and code) that will be shared by a set of objects.
- Each object of a given class contains the structure and behavior defined by the class.
- *Thus, a class is a logical construct; an object has physical reality.*

Encapsulation

- When you create a class, you will specify the code and data that constitute that class.
- The data defined by the class are referred to as *member variables* or *instance variables*.
- The code that operates on that data is referred to as *member methods* or just *methods*.

Encapsulation- A Class Representation



Public instance variables
(not recommended)



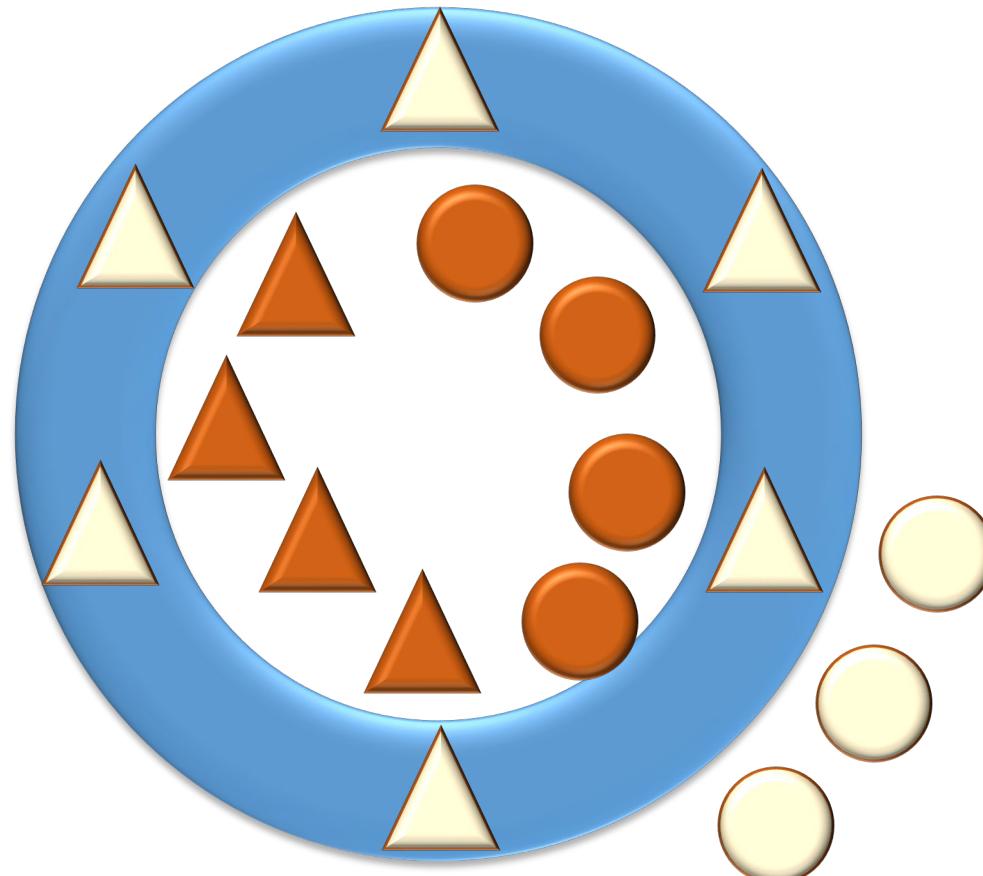
Public methods



Private methods



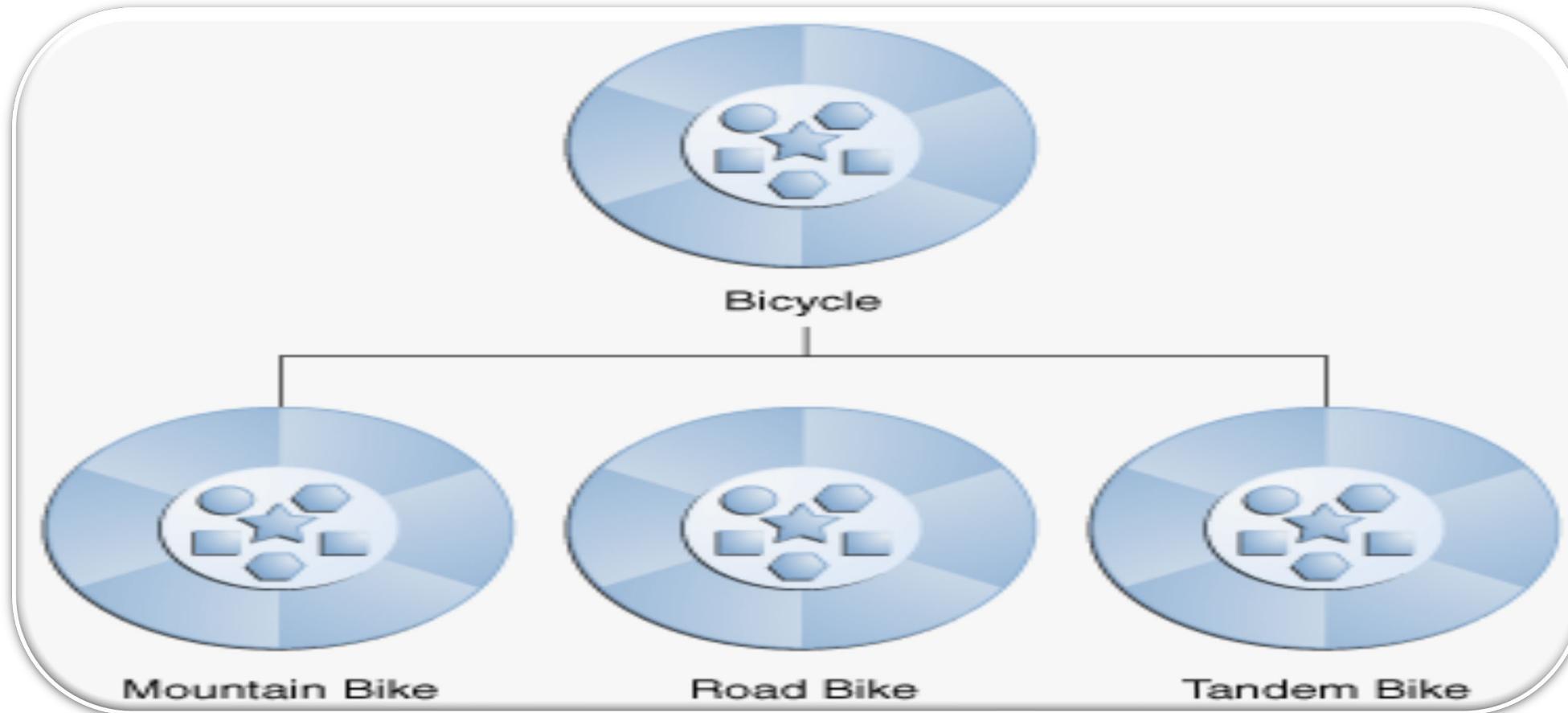
Private instance variables



Inheritance

- Inheritance is the process by which one object acquires the properties of another object.
- This is important because it supports the concept of hierarchical classification.
- Without the use of hierarchies, each object would need to define all of its characteristics explicitly.
- By use of inheritance, an object need only define those qualities that make it unique within its class. It can inherit its general attributes from its parent.

Inheritance

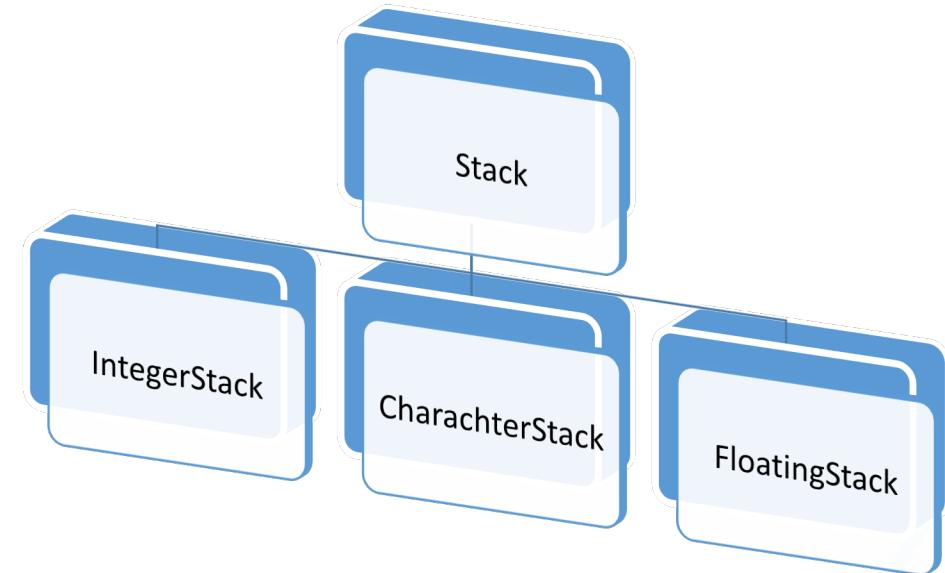


Polymorphism

- Polymorphism (from Greek, meaning “many forms”) is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation.
- More generally, the concept of polymorphism is often expressed by the phrase “*one interface, multiple methods.*” This means that it is possible to design a generic interface to a group of related activities.
- Consider a stack (which is a ***last-in, first-out*** list). You might have a program that requires three types of stacks. One stack is used for integer values, one for floating-point values, and one for characters.

Polymorphism

- It is the compiler's job to select the specific action (that is, method) as it applies to each situation.
- You, the programmer, do not need to make this selection manually. You need only remember and utilize the general interface.
- The specific action would be determined depending on the calling object



First Java Program

```
/*
This is a simple Java program.
Call this file "Example.java".
*/
class Example {
// Your program begins with a call to main().
public static void main(String args[]) {
System.out.println("This is a simple Java program.");
}
}
```

First Java Program

- For this example, the name of the source file should be **Example.java**.
- In Java, a source file is officially called a compilation unit.
- It is a text file that contains (among other things) one or more class definitions.
- The Java compiler requires that a source file use the .java filename extension.

Compiling the Program

- To compile the Example program, execute the compiler, *javac*, specifying the name of the source file on the command line, as shown here:

```
C:\>javac Example.java
```

- The javac compiler creates a file called Example.class that contains the bytecode version of the program.
- As discussed earlier, the Java bytecode is the intermediate representation of your program that contains instructions the Java Virtual Machine will execute.

Running the Program

- To actually run the program, you must use the Java application launcher called java. To do so, pass the class name Example as a command-line argument, as shown here:

```
C:\>java Example
```

- When the program is run, the following output is displayed:

```
C:\>This is a simple Java program.
```

First Sample Program Examined

- The program begins with the following lines:

```
/*
 *This is a simple Java program.
 *Call this file "Example.java".
 */
```

- The Above lines are multiline comments that are considered as remarks for the developer, explaining the operation of the program to anyone who is reading its source code.

Java three Styles of Comments

- `/*`
Comment }
`*/`

Multiline Comment

- `//` Comment

Single Line Comment

- `/**`
Comment }
`*/`

Comments processed by Javadoc tool

First Sample Program Examined

- The next line of code in the program is shown here:

```
class Example {
```

- This line uses the keyword ***class*** to declare that a new class is being defined.
- **Example** is an identifier that is the name of the class.
- The entire class definition, including all of its members, will be between the opening curly brace **{}** and the closing curly brace **}**.****

First Sample Program Examined

- The next line of code is shown here:

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

- As a general rule, a Java program begins execution by calling **main()**.
- The public keyword is an access modifier, which allows the programmer to control the visibility of class members.
- When a class member is preceded by public, then that member may be accessed by code outside the class in which it is declared.
- In this case, **main()** must be declared as public, since it must be called by code outside of its class when the program is started.

First Sample Program Examined

- The keyword **static** allows **main()** to be called without having to instantiate a particular instance of the class.
- This is necessary since **main()** is called by the Java Virtual Machine before any objects are made.
- The keyword **void** simply tells the compiler that **main()** does not return a value.
- **String args[]** declares a parameter named args, which is an array of instances of the class **String**.
- In this case, args receives any command-line arguments present when the program is executed.

First Sample Program Examined

- The next line of code is shown here. Notice that it occurs inside **main()**.

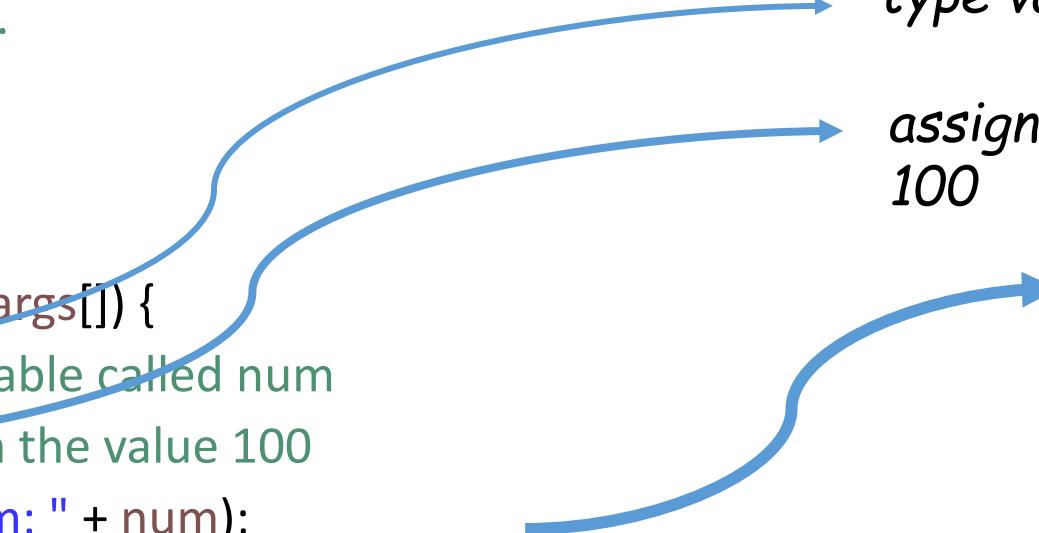
```
System.out.println("This is a simple Java  
program.");
```

- Output is actually accomplished by the built-in **println()** method.
- **System** is a predefined class that provides access to the system, and **out** is the output stream that is connected to the console.

Second Short Program

```
/*  
Here is another short example.  
Call this file "Example2.java".  
*/
```

```
class Example2 {  
public static void main(String args[]) {  
int num; // this declares a variable called num  
num = 100; // this assigns num the value 100  
System.out.println("This is num: " + num);  
num = num * 2;  
System.out.print("The value of num * 2 is ");  
System.out.println(num);  
}  
}
```



type var-name;
*assigns to num the value
100*
*the plus sign causes
the value of num to
be appended to the
string that precedes
it*

Two Control Statements- if statement

- The Java **if statement** works much like the **if statement** in any other language.
- It determines the flow of execution based on whether some condition is true or false. Its simplest form is shown here:
 - ***if(condition) statement;***
Where ***condition*** is a boolean expression that evaluates either ***true*** or ***false***

if(num < 100)

System.**out**.println("x is less than y");

- if num contains a value that is less than 100, the conditional expression is true, and println() will execute.

If Statement Example

```
/*
Demonstrate the if. Call this file "IfSample.java".
*/
class IfSample {
    public static void main(String args[]) {
        int x, y;
        x = 10; y = 20;
        if(x < y) System.out.println("x is less than y");
        x = x * 2;
        if(x == y) System.out.println("x now equal to y");
        x = x * 2;
        if(x > y) System.out.println("x now greater than y");
        // this won't display anything
        if(x == y) System.out.println("you won't see this");
    }
}
```

Two Control Statements- the for Loop

- Loop statements are an important part of nearly any programming language because they provide a way to repeatedly execute some task.
- The simplest form of the for loop is shown here:

for(initialization; condition; iteration) statement;

- The initialization portion of the loop sets a loop control variable to an initial value.
- The condition is a Boolean expression that tests the loop control variable.

Two Control Statements- the for Loop

- If the outcome of that test is true, statement executes and the for loop continues to iterate.
- If it is false, the loop terminates.
- The iteration expression determines how the loop control variable is changed each time the loop iterates.

For Loop Example

```
/*
Demonstrate the for loop.
Call this file "ForTest.java".
*/
class ForTest {
public static void main(String args[]) {
    int x;
    for(x = 0; x<10; x = x+1)
        System.out.println("This is x: " + x);
}
```

The Output:

This is x: 0
This is x: 1
This is x: 2
This is x: 3
This is x: 4
This is x: 5
This is x: 6
This is x: 7
This is x: 8
This is x: 9

Using Blocks of Code

- Java allows two or more statements to be grouped into ***blocks of code***, also called ***code blocks***.
- This is done by enclosing the statements between opening and closing curly braces.
- Once a block of code has been created, it becomes a logical unit that can be used any place that a single statement can.
- For example, a block can be a target for Java's if and for statements.

Block of Code Example

```
/*
```

Demonstrate a block of code.

Call this file "BlockTest.java"

```
*/
```

```
class BlockTest {
```

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

```
int x, y;
```

```
y = 20;
```

```
// the target of this loop is a block
```

```
for(x = 0; x<10; x++) {
```

```
System.out.println("This is x: " + x);
```

```
System.out.println("This is y: " + y);
```

```
y = y - 2;
```

```
} }
```

3-Dec-23

This is x: 0

This is y: 20

This is x: 1

This is y: 18

This is x: 2

This is y: 16

This is x: 3

This is y: 14

This is x: 4

This is y: 12

This is x: 5

This is y: 10

This is x: 6

This is y: 8

This is x: 7

This is y: 6

This is x: 8

This is y: 4

This is x: 9

This is y: 2

Lexical Issues- Atomic Elements of Java

- Java programs are a collection of whitespace, identifiers, literals, comments, operators, separators, and keywords.
- **Whitespace**
 - Java is a free-form language. This means that you do not need to follow any special indentation rules.

Lexical Issues- Atomic Elements of Java

- **Identifier**

- Identifiers are used to name things, such as classes, variables, and methods. An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters.
- They must not begin with a number, so as not to be confused with a numeric literal. Again, Java is case-sensitive, so **VALUE** is a different identifier than **Value**.

Valid identifiers	AvgTemp	count	a4	\$test	this_is_ok
Invalid identifiers	2count	high-temp	Not/ok	new Value	

NOTE: Beginning with JDK 9, the underscore cannot be used by itself as an identifier.

Lexical Issues- Atomic Elements of Java

- **Literals**
 - A constant value in Java is created by using a literal representation of it. For example, here are some literals:

100	98.6	'X', '\u03c0'	"This is a test"
integral literal	floating-point value	Character Literal	String Literal

Lexical Issues- Atomic Elements of Java

- **Separators**

- In Java, there are a few characters that are used as separators. The most commonly used separator in Java is the semicolon. As you have seen, it is often used to terminate statements.
- The separators are shown in the following table:

Lexical Issues- Atomic Elements of Java

Symbol	Name	Purpose
()	Parentheses	Used to contain lists of parameters in method definition and invocation
{ }	Braces	Used to contain the value of automatically initialized arrays. Also used to define a block of code, for classes, methods and local scopes
[]	Brackets	Used to declare array types. Also used when dereferencing array values
;	Semicolon	Terminates statements

Lexical Issues- Atomic Elements of Java

Symbol	Name	Purpose
,	Comma	Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a for statement
.	Period	Used to separate package names from subpackages and classes. Also used to separate a variable or method from a reference variable
::	Colons	Used to create a method or constructor reference
...	Ellipses	Indicates a variable-arity parameter
@	Ampersand	Begins an annotation

Lexical Issues- Atomic Elements of Java

- **The Java Keywords**

- There are 61 keywords currently defined in the Java language
- These keywords, combined with the syntax of the operators and separators, form the foundation of the Java language.
- In general, these keywords cannot be used as identifiers, meaning that they cannot be used as names for a variable, class, or method.
- The exceptions to this rule are the new context-sensitive keywords added by JDK 9 to support modules.
- The keywords **const** and **goto** are reserved but not used.
- In addition to the keywords, Java reserves the following: **true**, **false**, and **null**.

Lexical Issues- Atomic Elements of Java

- The Java Keywords

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

Lab Exercises

- Create simple applications that performs the following actions:
 - Display a message to the command prompt.
 - Receives an input and checks for its value and prints it back.
 - Receives two inputs a number and a string and prints the string on different lines according to the given number.

A.	The History and Evolution of Java	2
B.	An Overview of Java	21
C.	Data Types, Variables and Arrays	64
D.	Operators, Control Statements and String Handling	98
E.	Packages and Interfaces	163
F.	Wrapper Classes, Enumeration, Autoboxing and Annotations	216
G.	Exception Handling	256
H.	Generics and Lambda Expressions	299
I.	Java Stream API	363
J.	Java Collections	430
K.	Java Platform Module System	522

Lesson 2

Data Types, Variables, and Arrays

Introduction

- As with all modern programming languages, Java supports several types of datatypes. You may use these datatypes to declare variables and to create arrays.
- ***Java Is a Strongly Typed Language***
 - Every variable has a type, every expression has a type, and every type is strictly defined.
 - All assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility.
 - The Java compiler checks all expressions and parameters to ensure that the types are compatible.

The Primitive Types

- Java defines eight primitive types of data, that can be put in four groups:
- ***Integers***: This group includes **byte**, **short**, **int**, and **long**, which are for whole-valued signed numbers.
- ***Floating-point numbers***: This group includes **float** and **double**, which represent numbers with fractional precision.
- ***Characters***: This group includes **char**, which represents symbols in a character set, like letters and numbers.
- ***Boolean***: This group includes **boolean**, which is a special type for representing true/false values.

The Primitive Types

- The primitive types represent single values—not complex objects. Although Java is otherwise completely object-oriented, the primitive types are not.
- The reason for this is efficiency. Making the primitive types into objects would have degraded performance too much.
- The primitive types are defined to have an explicit range and mathematical behavior.
- Languages such as C and C++ allow the size of an integer to vary based upon the dictates of the execution environment.

Integer Types

Name	Width (in bits)	Range
long	64	-2^{63} to $2^{63} - 1$
int	32	-2^{31} to $2^{31} - 1$
short	16	-2^{15} to $2^{15} - 1$
byte	8	-2^7 to $2^7 - 1$

variable declaration examples:
 byte b , c;
 short s;
 short t;

Floating-Point Types

- Floating-point numbers, also known as real numbers, are used when evaluating expressions that require fractional precision.
- Java implements the standard (IEEE-754) set of floating-point types and operators.
- There are two kinds of floating-point types, float and double their width and ranges are shown here:

Name	Width in Bits	Approximate Range
double	64	4.9 e-324 to 1.8 e+308
float	32	1.4 e-45 to 3.4 e+38

Characters

- In Java, the data type used to store characters is **char**.
- Java uses Unicode to represent characters.
- Unicode defines a fully international character set that can represent all of the characters found in all human languages.
- At the time of Java's creation, Unicode required 16 bits. Thus, in Java char is a 16-bit type.
- The standard set of characters known as ASCII still ranges from 0 to 127 as always, and the extended 8-bit character set, ISO-Latin-1, ranges from 0 to 255.

Booleans

- Java has a primitive type, called boolean, for logical values.
- It can have only one of two possible values, **true** or **false**.
- This is the type returned by all relational operators, as in the case of `a < b`. **boolean** is also the type required by the conditional expressions that govern the control statements such as **if** and **for**.

Literals

- literals are the values that could be assigned to primitive data types or strings.
- literals are categorized according to the data types categories, **Integer, Floating-Point, Boolean, and Character Literals**

Integer Literals

```
int x=10;  
byte b=15;  
short s= 23;  
long l=15;
```

```
int x=012;  
byte b=0x15;  
short s= 0X23;  
long l=15L;
```

Beginning with JDK 7, you can also specify integer literals using binary or embed one or more underscores.

```
int x= 0b1010;  
int x=123_456_789;  
int x=0b1101_0101_0001_1010;
```

Floating-Point Literals

- Floating-point numbers represent decimal values with a fractional component.
- They can be expressed in either ***standard*** or ***scientific*** notation.

Standard Notation

```
double d=3.14159;  
double d1=0.6667;
```

Scientific Notation

```
double d=314159E-05;
```

- Floating-point literals in Java default to double precision.
- To specify a float literal, you must append an ***F*** or ***f*** to the constant.

Boolean Literals

- There are only two logical values that a boolean value can have, **true** and **false**.
- *The values of true and false do not convert into any numerical representation.*
- *The true literal in Java does not equal 1, nor does the false literal equal 0.*

Character Literals

- A literal character is represented inside a pair of single quotes.
- All of the visible ASCII characters can be directly entered inside the quotes, such as 'a', 'z', and '@'.
- For characters that are impossible to enter directly, there are several escape sequences that allow you to enter the character you need, such as '\'' for the single-quote character itself and '\n' for the newline character.

Character Literals

- For octal notation, use the backslash followed by the three digit number. For example, '**\141**' is the letter 'a'. For hexadecimal, you enter a backslash-u (\u), then exactly four hexadecimal digits.
- For example, '**\u0061**' is the ISO-Latin-1 'a' because the top byte is zero.

Escape Sequence	Description	Escape Sequence	Description
\ddd	Octal character	\uxxxx	Hexadecimal Unicode character
\'	Single quote	\"	Double quote
\\	Backslash	\r	Carriage return
\n	New Line	\f	Form feed
\t	Tab	\b	Backspace

Variables

- The variable is the basic unit of storage in a Java program.
- A variable is defined by the combination of an identifier, a type, and an optional initializer.
- In addition, all variables have a scope, which defines their visibility, and a lifetime.

Variables

- Declaring a Variable

- All variables must be declared before they can be used.
- The basic form of a variable declaration is shown here:

type identifier [= value][, identifier [= value] ...];

int a, b, c;

int d= 3, e, f=5;

byte z= 22;

double pi=3.14159;

char x='x';

Variables

- **Dynamic Initialization**
 - The preceding examples have used only constants as initializers
 - Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared.

double a=3.0, b=4.0;

double c= Math.sqrt (a*a + b*b);

Variables

- **The Scope and Lifetime of Variables**
 - Java allows variables to be declared within any block
 - A block defines a scope. Thus, each time you start a new block, you are creating a new scope.
 - A scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.
 - The two major scopes are those defined by a class and those defined by a method.
 - The scope defined by a method begins with its opening curly brace. However, if that method has parameters, they too are included within the method's scope.

Variables- Scope Example

```
//Demonstrate block scope.

class Scope {
    public static void main(String args[]) {
        int x; // known to all code within main
        x = 10;

        if(x == 10) { // start new scope
            int y = 20; // known only to this block
            // x and y both known here.
            System.out.println("x and y: " + x + " " + y);
            x = y * 2;

        } // y = 100; // Error! y not known here // x is still known here.

        System.out.println("x is " + x);
    }
}
```

Type Conversion and Casting

- It is fairly common to assign a value of one type to a variable of another type. If the two types are compatible, then Java will perform the conversion automatically.
- When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are met:
 - *The two types are compatible.*
 - *The destination type is larger than the source type.*
- When these two conditions are met, a **widening conversion** takes place.

Type Conversion and Casting

- For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other.
- However, there are no automatic conversions from the numeric types to ***char*** or ***boolean***. Also, ***char*** and ***boolean*** are not compatible with each other.
- Java also performs an automatic type conversion when storing a literal integer constant into variables of type ***byte***, ***short***, ***long***, or ***char***

Casting Incompatible Types

- What if you want to assign an int value to a byte variable?
- This conversion will not be performed automatically, because a **byte** is smaller than an **int**.
- This kind of conversion is sometimes called a narrowing conversion, since you are explicitly making the value narrower so that it will fit into the target type.
- To create a conversion between two incompatible types, you must use a cast. A cast is simply an explicit type conversion. It has this general form: **(target-type) value**

```
int a;  
byte b;  
b=(byte) a
```

Automatic Type Promotion in Expressions

- In addition to assignments, there is another place where certain type conversions may occur: in expressions.
- In an expression, the precision required of an intermediate value will sometimes exceed the range of either operand. For example, examine the following expression:

byte a=40;

byte b=50;

byte c =100;

int d=a * b / c;

Arrays

- An array is a group of like-typed variables that are referred to by a common name.
- Arrays of any type can be created and may have one or more dimensions.
- A specific element in an array is accessed by its index.
- Arrays offer a convenient means of grouping related information.

One-Dimensional Arrays

- A one-dimensional array is, essentially, a list of like-typed variables.
- To create an array, you first must create an array variable of the desired type.
- The general form of a one-dimensional array declaration is
type var-name[];  **int month_days [];**
- To link ***month_days*** with an actual, physical array of integers, you must allocate one using ***new*** and assign it to ***month_days***.
- ***new*** is a special operator that allocates memory.

One-Dimensional Arrays

- The general form of ***new*** as it applies to one dimensional arrays appears as follows:

array-var = new type [size];



month_days=new int [12];

- After this statement executes, ***month_days*** will refer to an array of 12 integers.
- Further, all elements in the array will be initialized to zero (The default value of the underlying data type).

One Dimensional Array Example

```
//Demonstrate a one-dimensional array.

class ArrayTest {
    public static void main(String args[]) {
        int month_days[];
        month_days = new int[12];
        month_days[0] = 31; month_days[1] = 28;
        month_days[2] = 31; month_days[3] = 30;
        month_days[4] = 31; month_days[5] = 30;
        month_days[6] = 31; month_days[7] = 31;
        month_days[8] = 30; month_days[9] = 31;
        month_days[10] = 30; month_days[11] = 31;
        System.out.println("April has " + month_days[3] + " days.");
    }
}
```

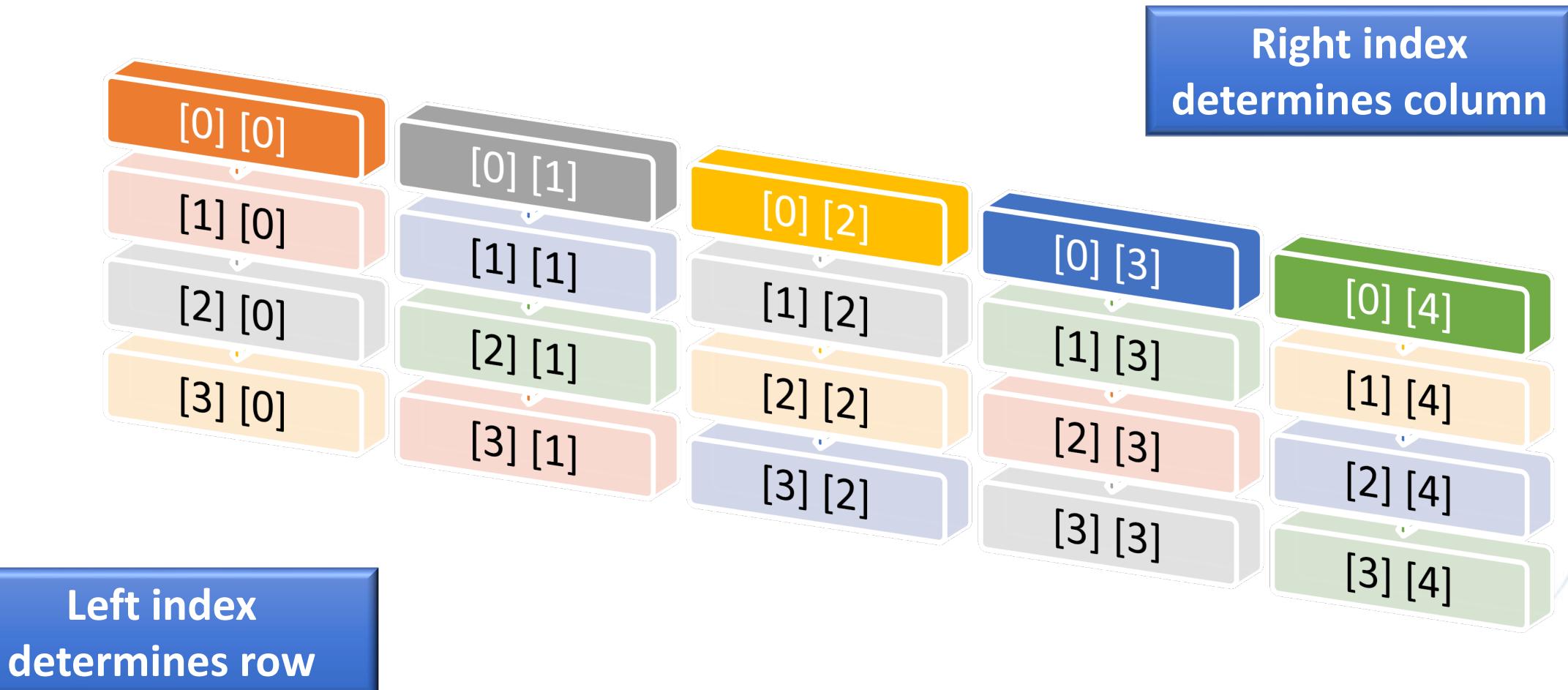
Multidimensional Arrays

- *Multidimensional* arrays are implemented as arrays of arrays.
- To declare a multidimensional array variable, specify each additional index using another set of square brackets.

int twoD[][] = new int [4][5]

- This allocates a 4 by 5 array and assigns it to twoD. Internally, this matrix is implemented as an array of arrays of int.
- Conceptually, this array will look like the one shown in the following figure

2-Dimensional Array Conceptual View



Lab Exercise

- Develop an application that extracts the minimum and maximum of the elements of an array of 1000 element and compute the search running time.
- Rewrite the application to implement the binary search algorithm and compute the search running time.

Hint: Use `System.currentTimeMillis()` or `System.nanoTime ()`.

Lesson 3

Operators, Control Statements, and String Handling

A.	The History and Evolution of Java	2
B.	An Overview of Java	21
C.	Data Types, Variables and Arrays	64
D.	Operators, Control Statements and String Handling	98
E.	Packages and Interfaces	163
F.	Wrapper Classes, Enumeration, Autoboxing and Annotations	216
G.	Exception Handling	256
H.	Generics and Lambda Expressions	299
I.	Java Stream API	363
J.	Java Collections	430
K.	Java Platform Module System	522

Operators

- Arithmetic Operators
- Bitwise Operators
- Relational Operators
- Boolean Logical Operators
- The Ternary Operator (?)
- **instanceof** Operator
- The arrow Operator (->)

Arithmetic Operators

Operator	Result
+	Addition (also unary plus)
-	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition Assignment
-=	Subtraction Assignment
*=	Multiplication assignment
/=	Division Assignment
%=	Modulus Assignment
--	Decrement

- The operands of the arithmetic operators must be of a numeric type.
- Remember that when the division operator is applied to an integer type, there will be no fractional component attached to the result.

The Bitwise Operators

- Java defines several bitwise operators that can be applied to the integer types: **long, int, short, char, and byte**.
- These operators act upon the individual bits of their operands. They are summarized in the following table:

The Bitwise Operators

Operator	Result	Operator	Result
<code>~</code>	Bitwise unary Not	<code>&=</code>	Bitwise AND Assignment
<code>&</code>	Bitwise AND	<code> =</code>	Bitwise OR Assignment
<code> </code>	Bitwise OR	<code>^=</code>	Bitwise Exclusive OR Assignment
<code>^</code>	Bitwise Exclusive OR	<code>>>=</code>	Shift Right Assignment
<code>>></code>	Shift Right	<code>>>>=</code>	Shift Right zero fill Assignment
<code>>>></code>	Shift Right zero fill	<code><<=</code>	Shift left Assignment
<code><<</code>	Shift left		

The Bitwise Operators

- Assume if $a = 60$; and $b = 13$;

$a = 0011\ 1100$

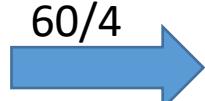
$b = 0000\ 1101$

$$=0*2^0+0*2^1+1*2^2+1*2^3\\+1*2^4+1*2^5+0*2^6+0*2^7$$

$a \& b = 0000\ 1100$

$a | b = 0011\ 1101$

$a ^ b = 0011\ 0001$

$a << 2$		11110000		$60 * 4$	240
$a >> 2$		00001111		$60 / 4$	15
$a >>> 2$		00001111			15
$\sim a$		11000011			-61

Relational Operators

- The relational operators determine the relationship that one operand has to the other
- They determine equality and ordering.
- The relational operators are shown here:

Operator	Result	Operator	Result
<code>==</code>	Equal to	<code><</code>	Less than
<code>!=</code>	Not equal to	<code>>=</code>	Greater than or equal to
<code>></code>	Greater than	<code><=</code>	Less than or equal to

- The outcome of these operations is a boolean value.
- The relational operators are most frequently used in the expressions that control the if statement and the various loop statements.

Boolean Logical Operators

- The Boolean logical operators shown below, operate only on boolean operands.
- All of the binary logical operators combine two boolean values to form a resultant boolean value.

Operator	Result	Operator	Result
&	Logical AND	&&	Short-circuit AND
	Logical OR		Short-circuit OR
^	Logical XOR (exclusive OR)	&=	AND assignment
!	Logical Unary Not	=	OR assignment
==	Equal to	!=	Not equal to

Boolean Logical Operators

A				B
False				False
True				False
False				True
True				True

Boolean Operators Example

```
//Demonstrate the boolean logical operators.

class BoolLogic {
    public static void main(String args[]) {
        boolean a = true;
        boolean b = false;
        boolean c = a | b;
        boolean d = a & b;
        boolean e = a ^ b;
        boolean f = (!a & b) | (a & !b);
        boolean g = !a;

        System.out.println("a = " + a); System.out.println("b = " + b);
        System.out.println("a|b = " + c); System.out.println("a&b = " + d);
        System.out.println("a^b = " + e); System.out.println("!a&b|a&!b = " + f);
        System.out.println("!a = " + g);
    }
}
```

The Ternary Operator (?)

- Java includes a special ternary (three-way) operator that can replace certain types of if-then-else statements.
- The ? has this general form:

expression1 ? expression2 : expression3

- Here, expression1 can be any expression that evaluates to a boolean value. If expression1 is true, then expression2 is evaluated; otherwise, expression3 is evaluated.

double ratio= total == 0 ? 0 : num / total

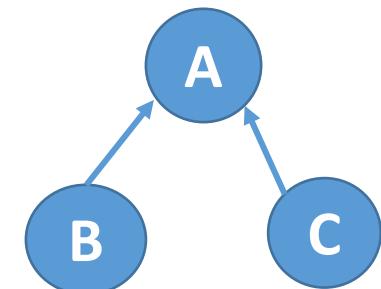
instanceof Operator

- knowing the type of an object during runtime is useful.
- For example, you might have one thread of execution that generates various types of objects, and another thread that processes these objects.
- In this situation, it might be useful for the processing thread to know the type of each object when it receives it.
- Another situation in which knowledge of an object's type at run time is important involves casting. In Java, an invalid cast causes a run-time error.

instanceof Operator

- For example, a superclass called A can produce two subclasses, called B and C. Thus, casting a B object into type A or casting a C object into type A is legal, but casting a B object into type C (or vice versa) isn't legal.
- Java provides the run-time operator instanceof to answer this question. The instanceof operator has this general form:

objref instanceof type



- Here, objref is a reference to an instance of a class, and type is a class type.

instanceof example

```
class A {  
    int i, j;  
}  
  
class B {  
    int i, j;  
}  
  
class C extends A {  
    int k;  
}  
  
class D extends A {  
    int k;  
}
```

instanceof example

```

class InstanceOf {
    public static void main(String args[]) {
        A a = new A();
        B b = new B();
        C c = new C();
        D d = new D();

        if(a instanceof A)
            System.out.println("a is instance of A");
        if(b instanceof B)
            System.out.println("b is instance of B");
        if(c instanceof C)
            System.out.println("c is instance of C");
        if(c instanceof A)
            System.out.println("c can be cast to A");
        if(a instanceof C)
            System.out.println("a can be cast to C");
        System.out.println();
    }
}

```

Output:

a is instance of A
 b is instance of B
 c is instance of C
 c can be cast to A

instanceof example

```
// compare types of derived types
A ob;
ob = d; // A reference to d
System.out.println("ob now refers to d");
if(ob instanceof D)
System.out.println("ob is instance of D");
System.out.println();
ob = c; // A reference to c
System.out.println("ob now refers to c");
if(ob instanceof D)
System.out.println("ob can be cast to D");
else
System.out.println("ob cannot be cast to D");
if(ob instanceof A)
```

Output:
ob now refers to d
ob is instance of D

Output:
ob now refers to c
ob cannot be cast to D

instanceof example

```
System.out.println("ob can be cast to A");
System.out.println();
// all objects can be cast to Object
if(a instanceof Object)
    System.out.println("a may be cast to Object");
if(b instanceof Object)
    System.out.println("b may be cast to Object");
if(c instanceof Object)
    System.out.println("c may be cast to Object");
if(d instanceof Object)
    System.out.println("d may be cast to Object");
}
```

Output:
ob can be cast to A

Output:
a may be cast to Object
b may be cast to Object
c may be cast to Object
d may be cast to Object

The arrow Operator (->)

- As of JDK 8, a new feature has been added to Java that profoundly enhances the expressive power of the language.
- This feature is the *lambda expression*.
- Not only do lambda expressions add new syntax elements to the language, they also streamline the way that certain common constructs are implemented.
- The addition of lambda expressions have also provided the catalyst for other new Java features (*Default Methods-Method Reference*).

Lambda Expression

A *lambda expression* is, essentially, an anonymous (that is, unnamed) method.

However, this method is not executed on its own. Instead, it is used to implement a method defined by a functional interface.

Thus, a lambda expression results in a form of anonymous class.

Lambda expressions are also commonly referred to as *closures*.



A *functional interface* is an interface that contains one and only one abstract method

Lambda Expressions

The lambda expression introduces a new syntax element and operator into the Java language. The new operator, sometimes referred to as the *lambda operator* or the *arrow operator*, is \rightarrow .

It divides a lambda expression into two parts.

The left side specifies any parameters required by the lambda expression. On the right side is the *lambda body*, which specifies the actions of the lambda expression.

Single Lambda Expressions(Examples)

() -> 98.6



- It evaluates to a constant value.
- This lambda expression takes no parameters, thus the parameter list is empty.
- It returns the constant value 98.6.
- Therefore, it is similar to the following method:
 - double myMeth() { return 98.6; }
- Of course the method defined by a lambda expression does not have a name

()-> Math.random() *100



This lambda expression obtains a pseudo-random value from **Math.random()**, multiplies it by 100, and returns the result. It, too, does not require a parameter.

Single Lambda Expressions(Examples)

When a lambda expression requires a parameter, it is specified in the parameter list on the left side of the lambda operator. Here is a simple example:

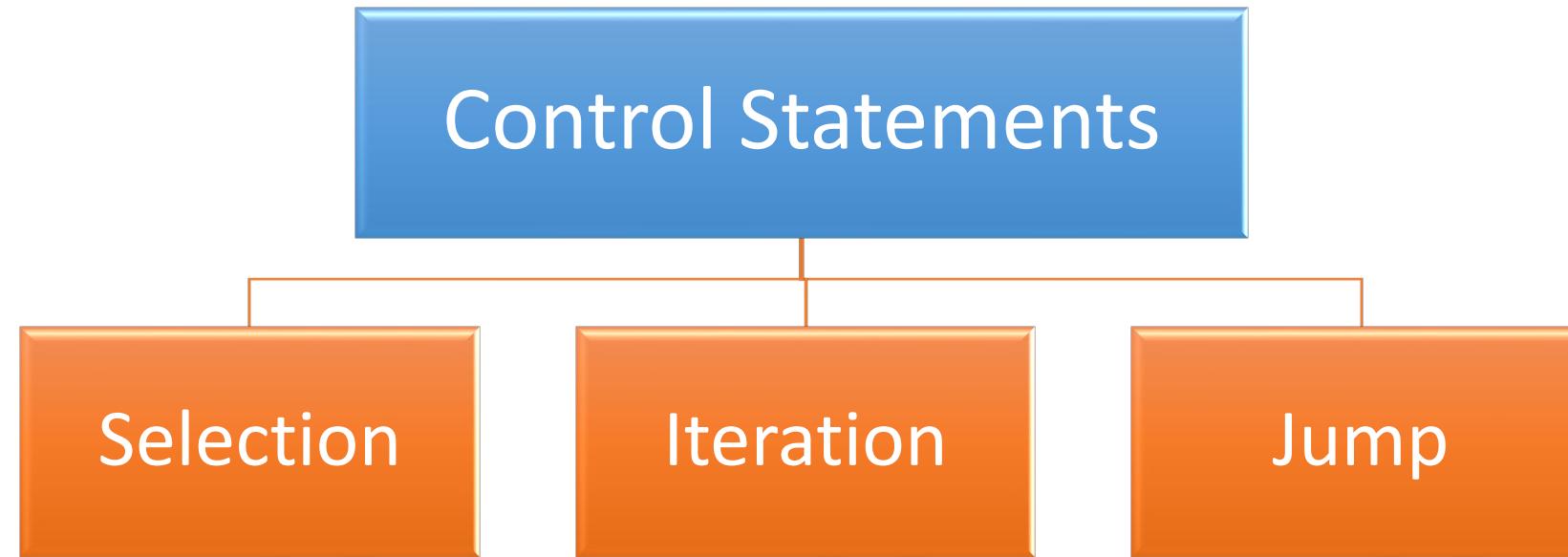
(n) -> 1.0 / n

- This lambda expression returns the reciprocal of the value of n
- If n is 4.0 then the reciprocal is 0.25
- The type of n can be inferred so you don't need to explicitly specify it.
- Like a named method, a lambda expression can specify as many parameters as needed

(n) -> (n % 2)==0

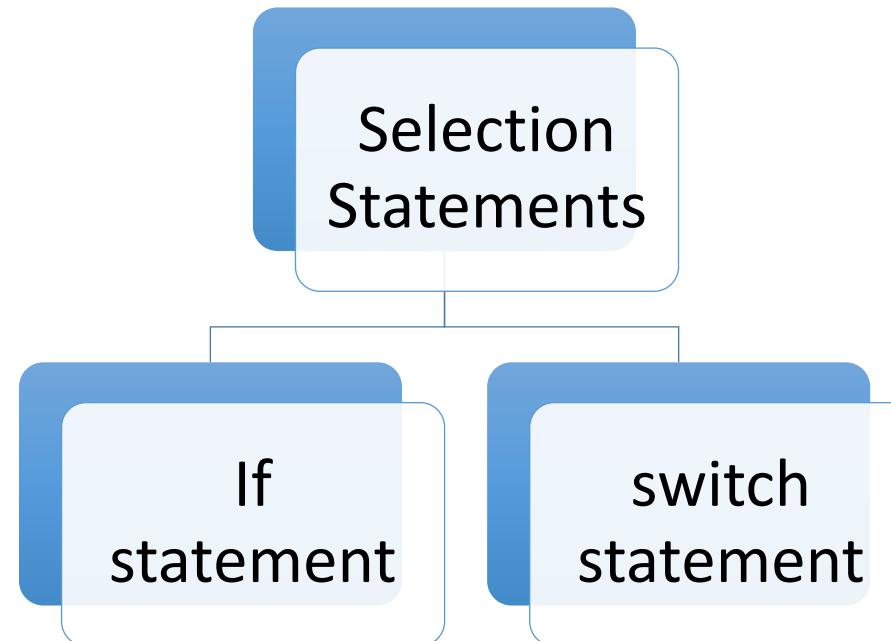
- Any valid type can be used as the return type of a lambda expression.
- The above lambda expression return **boolean**
- When a lambda expression has only one parameter, it is not necessary to surround the parameter name with parentheses when it is specified on the left side.

Control Statements in Java

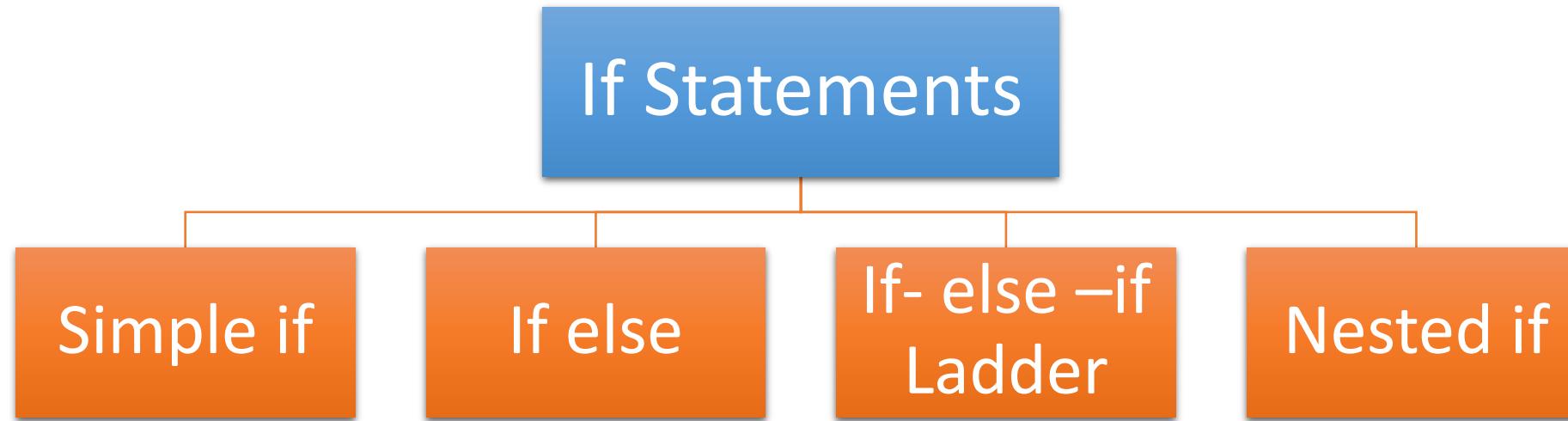


Selection Statements

- Selection Statements are also called Decision Making Statements.



If Statements



Simple if

Syntax :

```
if (condition)
{
    statement1;
}
```

```
if(x < y) {
    System.out.println("x is Less than y");
}
```

Purpose: The statements will be evaluated if the value of the condition is true.

If else

Syntax :

```
if (condition)
{
    statement1;
}
else
{
    statement2;
}
```

```
if(x < y) {
    System.out.println("x is Less than y");
}else {
    System.out.println("x is either equals
y or greater than y");
}
```

Purpose: The statement 1 is evaluated if the value of the condition is true otherwise statement 2 is true.

If-else-if Ladder

Syntax :

```
if(condition)
    statements;
else if(condition)
    statements;
else if(condition)
    statements;

...
...
else
    statements;
```

If-else-if example

```
class IfElse {  
    public static void main(String args[]) {  
        int month = 4; // April  
        String season;  
  
        if(month == 12 || month == 1 || month == 2)  
            season = "Winter";  
        else if(month == 3 || month == 4 || month == 5)  
            season = "Spring";  
        else if(month == 6 || month == 7 || month == 8)  
            season = "Summer";  
        else if(month == 9 || month == 10 || month == 11)  
            season = "Autumn";  
        else  
            season = "Unknown Month";  
        System.out.println("April is in the " + season + ".");  
    }  
}
```

Nested if

- A nested if is an if statement that is the target of another if or else.
- Nested ifs are very common in programming.

Syntax :

```
if(condition){  
    if(condition) statements....  
    else  
        statements....  
}else{  
    if(condition)  
        statements....  
    else  
        statements....  
}
```

Example

```
class NestedIfDemo
{
public static void main(String args[])
{
int i = 10;

if (i == 10)
{
// First if statement
if (i < 15)
System.out.println("i is smaller than 15");
// Nested - if statement
// Will only be executed if statement above
// it is true
if (i < 12)
System.out.println("i is smaller than 12 too");
else
System.out.println("i is greater than 15");
}
}
}
```

switch case

Purpose: The statements N will be evaluated if the value of the expression matches the value N.

Syntax :

```
switch (expression)
{
    case value 1 :
        statement 1 ;
        break;
    case value 2 :
        statement 2 ;
        break;
    ...
    case value N : statement N ; break;
    default :
        statements ; break;
}
```

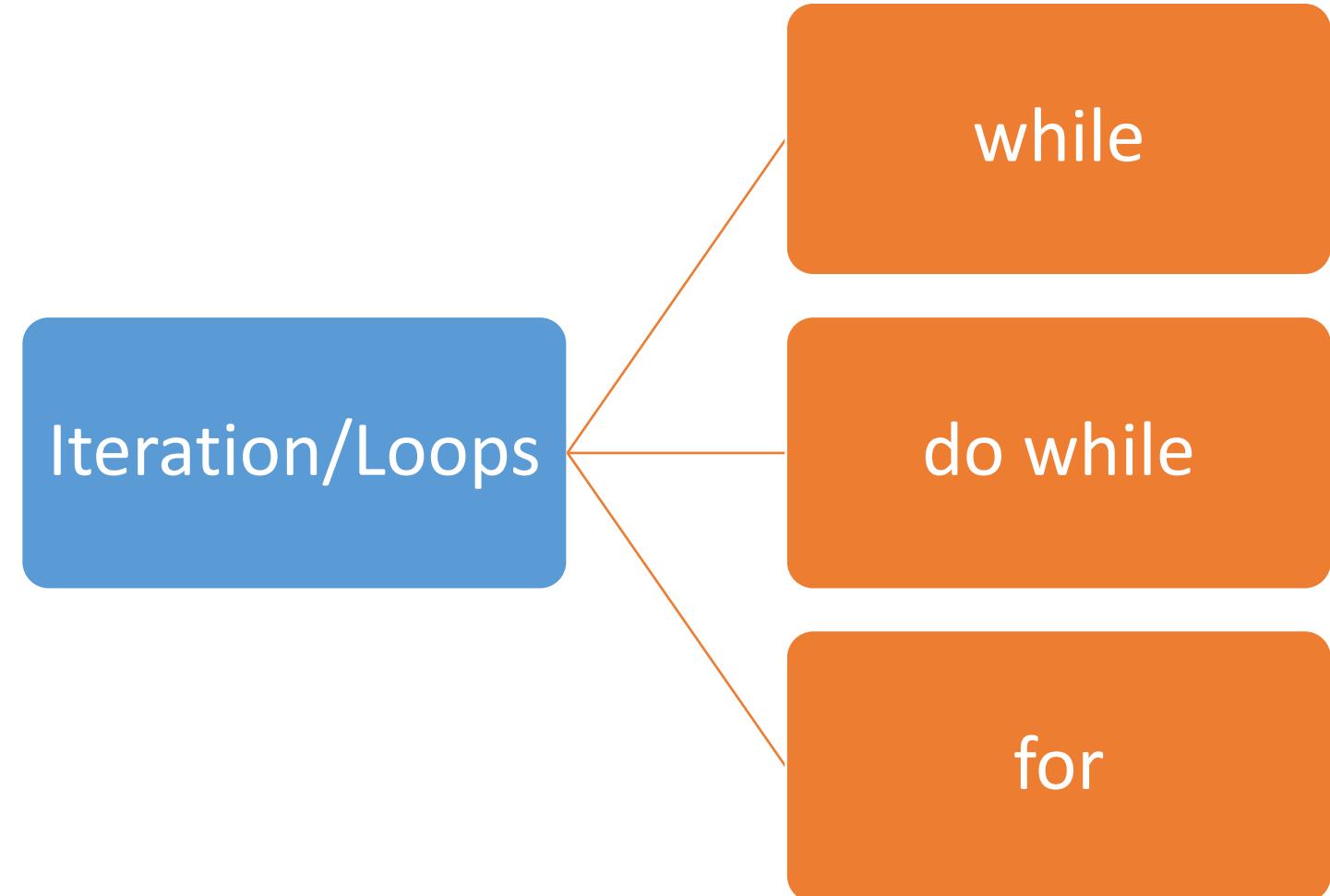
Example

```
class SwitchCaseDemo
{
public static void main(String args[])
{
    int i = 9;
    switch (i)
    {
        case 0:
            System.out.println("i is zero.");
            break;
        case 1:
            System.out.println("i is one.");
            break;
        case 2:
            System.out.println("i is two.");
            break;
        default:
            System.out.println("i is greater than 2.");
    }
}
```

Iteration Statements

Each loop has four types of statements :

- ***Initialization***
- ***Condition checking***
- ***Execution***
- ***Increment / Decrement***



while loop

- **Syntax**

```
initialization
while(final value)
{
    statements;
increment/decrement;
}
```

- **Purpose:** To evaluate the statements from initial value to final value with given increment/decrement.

```
int m=1;
while(m<=20){
    System.out.println(m);
    m=m+1;
}
```

do while loop

- **Syntax**

initialization

do{

statements;

increment/decrement;

}

while(final value);

- **Purpose:** To evaluate the statements from initial value to final value with given increment/decrement.

```
int m=1;  
do{  
    System.out.println(m);  
    m=m+1;  
} while(m<=20);
```

for loop

- **Syntax**

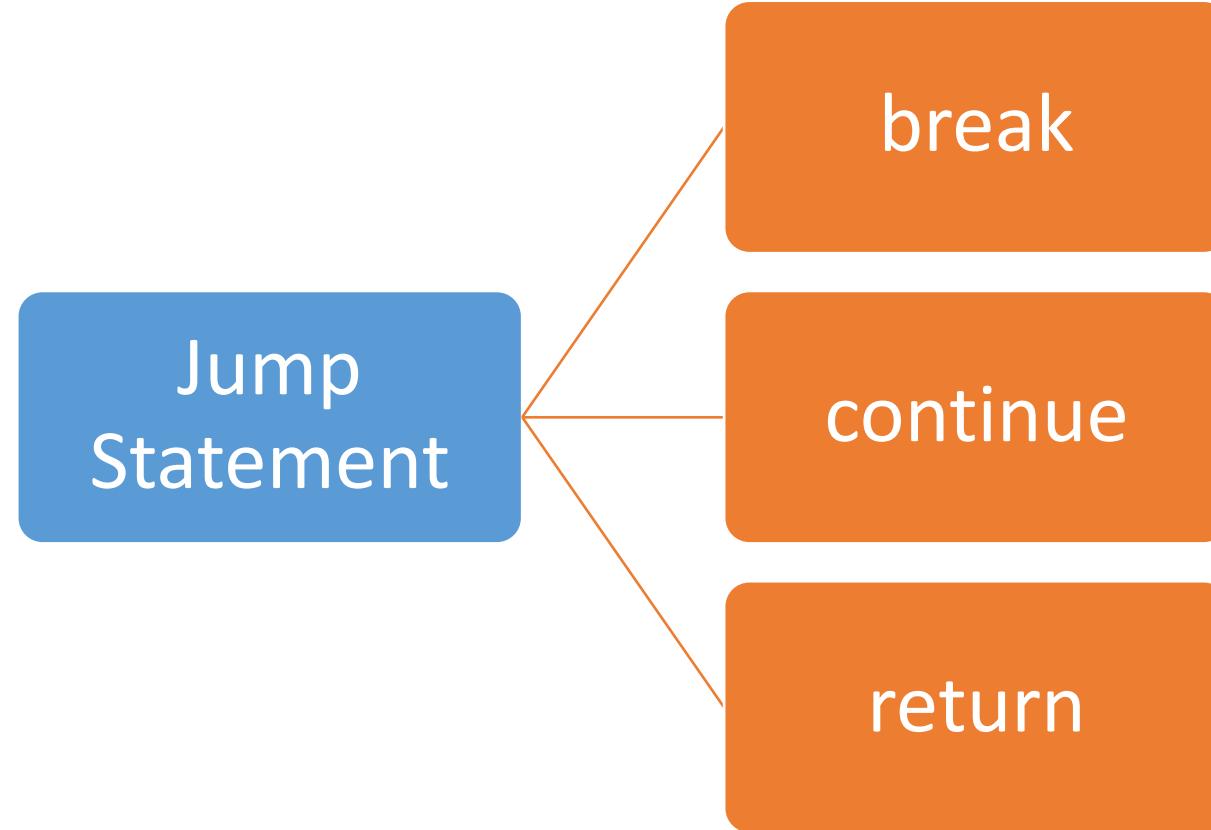
for(initialization; termination condition; step)

```
{  
    statements;  
}
```

- **Purpose:** To evaluate the statements from initial value to final value with given step.

```
for(int i=0; i<10; i++)  
{  
    System.out.println(i);  
}
```

Jump Statements



The break statement

- This statement is used to jump out of a loop.
- Break statement was previously used in switch – case statements.
- On encountering a break statement within a loop, the execution continues with the next statement outside the loop.
- The remaining statements which are after the break and within the loop are skipped.
- Break statement can also be used with the label of a statement.
- A statement can be labeled as follows.

statementName : SomeJavaStatement

When we use break statement along with label as:

break statementName;

Example

```
class BreakLabelDemo {  
    public static void main(String args[]) {  
        outer: for (int i = 0; i < 3; i++) {  
            System.out.print("Pass " + i + ":" );  
            for (int j = 0; j < 100; j++) {  
                if (j == 10)  
                    break outer; // exit both loops  
                System.out.print(j + " ");  
            }  
            System.out.println("This will not print");  
        }  
        System.out.println("Loops complete.");  
    }  
}
```

continue Statement

- This statement is used only within looping statements.
- When the continue statement is encountered, the next iteration starts.
- The remaining statements in the loop are skipped. The execution starts from the top of loop again.

Example

```
class ContinueLabel {  
    public static void main(String args[]) {  
outer: for (int i=0; i<10; i++) {  
        for(int j=0; j<10; j++) {  
            if(j > i) {  
                System.out.println();  
                continue outer;  
            }  
            System.out.print(" " + (i * j));  
        }  
        System.out.println();  
    }  
}
```

The return Statement

- The last control statement is return. The return statement is used to explicitly return from a method.
- That is, it causes program control to transfer back to the caller of the method.
- The return statement immediately terminates the method in which it is executed.

Example

```
//Demonstrate return.  
class ReturnDemo {  
public static void main(String args[]) {  
boolean t = true;  
System.out.println("Before the return.");  
if (t)  
return; // return to caller  
System.out.println("This won't execute.");  
}  
}
```

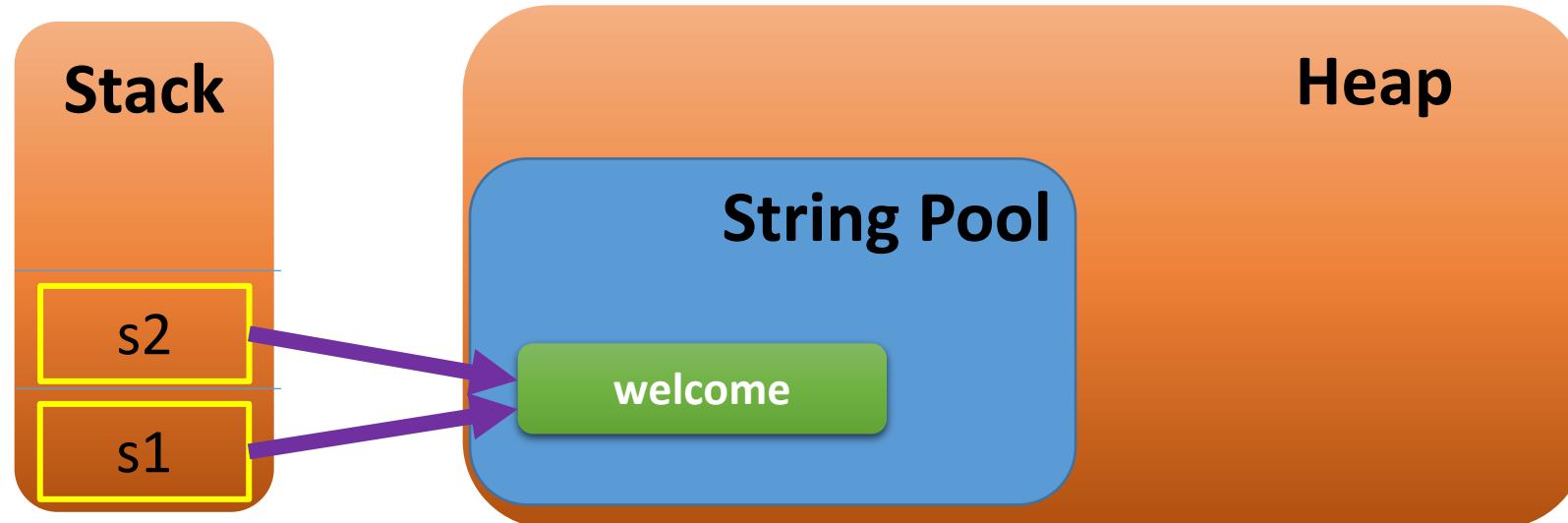
String Handling

The String Class (java.lang.String)

- String is a sequence of characters placed in double quote(“ ”).
- Strings are constant , their values cannot be changed in the same memory after they are created.

How to create String object

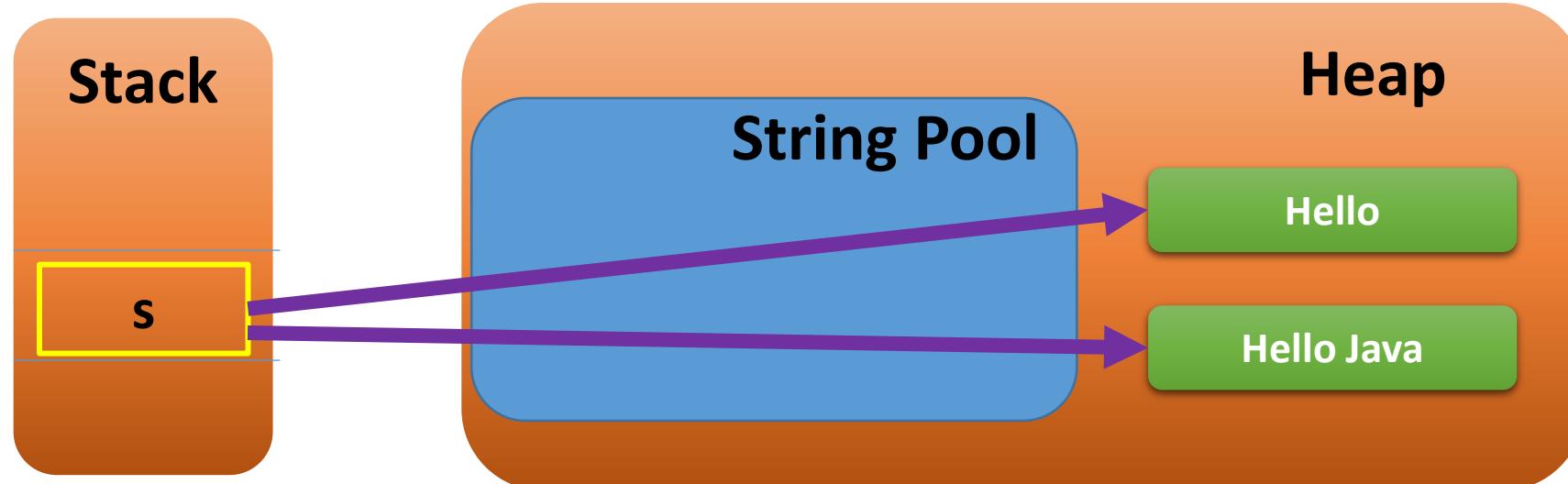
- There are two ways to create String object:
 - By string literal.
 - By new keyword.
- By string literal:
 - For Example: String s1="welcome";
 - String s2="welcome"; //no new object will be created



Create String By new keyword:-

- For Example:

```
String s=new String("Hello");  
s=new String("Hello Java");
```



Immutability

In java, string objects are immutable.

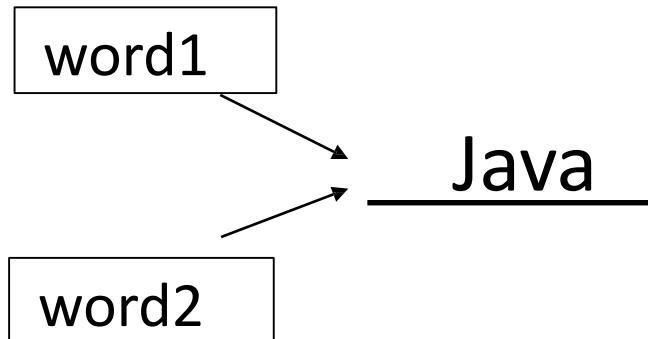
Immutable simply means unmodifiable or unchangeable.

Once string object is created its data or state can't be changed but a new string object is created.

Advantages Of Immutability

Use less memory:

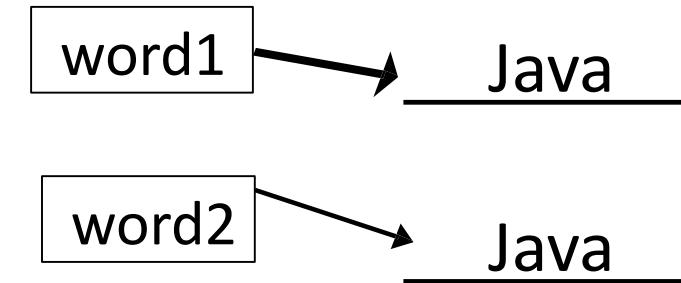
```
String word1 = "Java";  
String word2 = word1;
```



more efficient: saves
memory

3-Dec-23

```
String word1 = "Java";  
String word2 = new String("Java");
```



Less efficient: wastes
memory

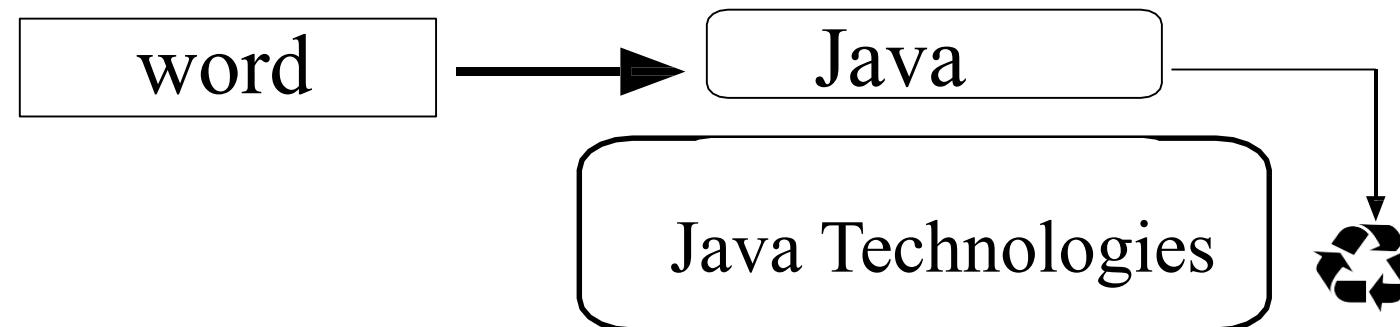
147

Disadvantages of Immutability

Less efficient — you need to create a new string and throw away the old one even for small changes.

Example:

```
String word = "Java";  
word= word.concat("Technologies");
```



Constructors

- **No Argument Constructors:**
 - No-argument constructor creates an empty String. Rarely used.
Example: **String empty=new String();**
- **Other Constructors:**
 - Most other constructors take an array as a parameter to create a String.

Example:

```
char[] letters = {'J', 'a', 'v', 'a'};  
String word = new String(letters); // "Java"
```

String Methods

- **substring(int begin):**

- Returns substring from begin index to end of the String.

Example:

```
String s="helping";
System.out.println(s.substring(2));//lping
```

- **equals():**

- To perform content comparision where case is important.

- Example:

```
String s="java";
```

```
System.out.println(s.equals("java")); //true
```

- **concat():** Adding two strings we use this method

Example:

```
String s="test";
s= s.concat("software");
System.out.println(s);
// testsoftware
```

length() , charAt()

int length();

- Returns the number of characters in the string
- Returns the char at position i.

char charAt(i);

Character positions in strings are numbered starting from 0 – just like arrays.

Returns:

“Problem”.length();

→ 7

“Window”. charAt (2);

→ 'n'

New features of String

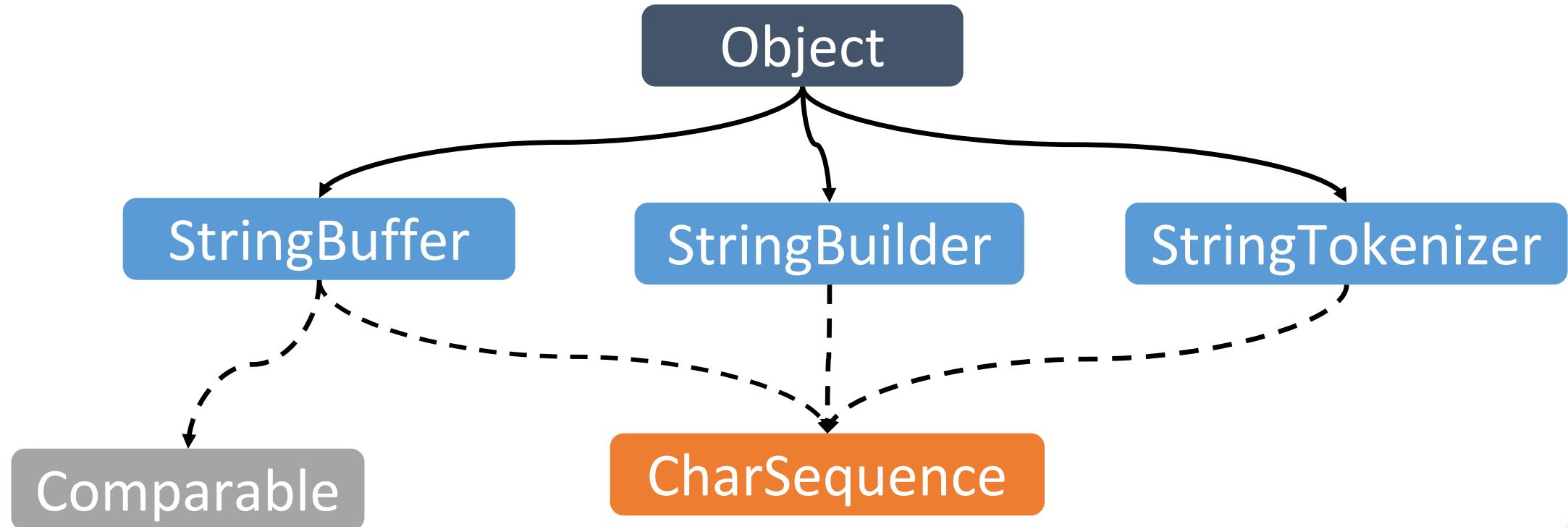
- In JDK 1.7 (Java 7):
 - Java 7 extended the capability of switch case to use Strings also, earlier java versions doesn't support this.
 - If you are implementing conditional flow for Strings, you can use if-else conditions and you can use switch case if you are using Java 7 or higher versions.

StringBuffer, StringBuilder StringTokenizer

StringBuffer, StringBuilder and StringTokenizer

- **Limitation of String in Java ?**
- **What is mutable string?**
 - A string that can be modified or changed is known as mutable string. StringBuffer and StringBuilder classes are used for creating mutable string.
- **Differences between String and StringBuffer in java?**
 - Main difference between String and StringBuffer is String is immutable while StringBuffer is mutable

Relations between StringBuffer, StringBuilder and StringTokenizer:



All These classes are final and implement Serializable.

StringBuffer

- StringBuffer is synchronized and allows us to mutate the string.
- StringBuffer has many utility methods to manipulate the string.
- This is more useful when using in a multithreaded environment.
- Always modified in same memory location.

StringBuffer

- **StringBuffer Constructors:**
 - public StringBuffer()
 - public StringBuffer(int capacity)
 - public StringBuffer(String s)
 - public StringBuffer(CharSequence cs)

Method Use In StringBuffer

- append
- insert
- delete
- reverse
- Replacing Character at given index

StringBuilder

- StringBuilder is the same as the StringBuffer class
- The StringBuilder class is not synchronized and hence in a single threaded environment, the overhead is less than using a StringBuffer.

StringTokenizer

- A *token* is a portion of a string that is separated from another portion of that string by one or more chosen characters (called *delimiters*).
- The *StringTokenizer* class contained in the *java.util package* can be used to break a string into separate tokens.
- This is particularly useful in those situations in which we want to read and process one token at a time;

Lab Exercise

- Given a sentence and a word, your task is that to count the number of occurrences of the given word in the string and print the number of occurrence of the word.
 1. Perform the above task using only methods of the String class (2 ways).
 2. Perform the above task using StringTokenizer class
- Count a group of words in a string using regular expressions

A.	The History and Evolution of Java	2
B.	An Overview of Java	21
C.	Data Types, Variables and Arrays	64
D.	Operators, Control Statements and String Handling	98
E.	Packages and Interfaces	163
F.	Wrapper Classes, Enumeration, Autoboxing and Annotations	216
G.	Exception Handling	256
H.	Generics and Lambda Expressions	299
I.	Java Stream API	363
J.	Java Collections	430
K.	Java Platform Module System	522

Lesson 4

Packages and Interfaces

Packages and interfaces

Packages

Packages

- In the preceding lessons, the name of each example class was taken from the same name space. This means that a unique name had to be used for each class to avoid name collisions.
- After a while, without some way to manage the name space, you could run out of convenient, descriptive names for individual classes.
- You also need some way to be assured that the name you choose for a class will be reasonably unique and not collide with class names chosen by other programmers.

Packages

- Java provides a mechanism for partitioning the class name space into more manageable chunks.
- This mechanism is the *package*. The *package* is both a naming and a visibility control mechanism.
- You can define classes inside a package that are not accessible by code outside that package.
- You can also define class members that are exposed only to other members of the same package.

Defining a Packages

- To create a package is quite easy: simply include a **package** command as the first statement in a Java source file.
- Any classes declared within that file will belong to the specified package.
- The **package** statement defines a name space in which classes are stored. If you omit the **package** statement, the class names are put into the default package, which has no name.
- This is the general form of the **package** statement:

package *pkg*;

pkg is the name of the package;

Defining a Packages

- example, the following statement creates a package called **mypackage**:

```
package mypackage;
```

- Java uses file system directories to store packages. For example, the **.class** files for any classes you declare to be part of **mypackage** must be stored in a directory called **mypackage**.
- The general form of a multileveled package statement is shown here:

package pkg1[pkg2[pkg3]]; → package a.b.c;

Finding Packages and CLASSPATH

- How does the Java run-time system know where to look for packages that you create?
 - **First**, by default, the Java run-time system uses the current working directory as its starting point.
 - **Second**, you can specify a directory path or paths by setting the **CLASSPATH** environmental variable.
 - **Third**, you can use the **-classpath** option with **java** and **javac** to specify the path to your classes.
- *It is useful to point out that, beginning with JDK 9, a package can be part of a module, and thus found on the module path.*

Packages and Member Access

- Packages add another dimension to access control. As you will see, Java provides many levels of protection to allow fine-grained control over the visibility of variables and methods within classes, subclasses, and packages.
- Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods.
- Packages act as containers for classes and other subordinate packages. Classes act as containers for data and code.

Packages and Member Access

- Java addresses four categories of visibility for class members:
 - Subclasses in the same package
 - Non-subclasses in the same package
 - Subclasses in different packages
 - Classes that are neither in the same package nor subclasses
- The three access modifiers, **private**, **public**, and **protected**, provide a variety of ways to produce the many levels of access required by these categories. The following table sums up the interactions.

Packages and Member Access

	private	No Modifier	protected	public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

NOTE: The modules feature added by JDK 9 can also affect accessibility.

Importing packages

- In a Java source file, **import** statements occur immediately following the **package** statement (if it exists) and before any class definitions. This is the general form of the **import** statement:

*import pkg1 [.pkg2].(classname | *);*

Example:

import java.util.Date;

import java.io.;*

Standard Java SE packages

- All of the standard Java SE classes included with Java begin with the name **java**.
- The basic language functions are stored in a package called **java.lang**.
- Normally, you have to import every package or class that you want to use, but since Java is useless without much of the functionality in **java.lang**, it is implicitly imported by the compiler for all programs.

Standard Java SE packages

Package Name	Description
java.lang	Contains language support classes (for e.g classes which defines primitive data types, math operations, etc.) . This package is automatically imported.
java.io	Contains classes for supporting input / output operations.
java.util	Contains utility classes which implement data structures like Linked List, Hash Table, Dictionary, etc and support for Date / Time operations.
java.applet	Contains classes for creating Applets.
java.awt	Contains classes for implementing the components of graphical user interface (like buttons, menus, etc.).
3-Dec-23 java.net	Contains classes for supporting networking operations.

Interfaces

Interfaces

- In OOP, it is sometimes helpful to define what a class must do but not how it will do it.
- An **abstract method** defines the **signature** for a method but provides no **implementation**.
- A **subclass** must provide its own **implementation** of each abstract method defined by its **superclass**.
- Thus, an **abstract method** specifies the **interface** to the method but not the **implementation**.
- In Java, you can fully separate a class' interface from its implementation by using the keyword **interface**.

Interfaces

- An *interface* is syntactically similar to an **abstract class**, in that you can specify one or more methods that have no body.
- Those methods must be **implemented by a class** in order for their actions to be defined.
- An *interface* specifies what **must be done**, but **not how to do it**.
- Once an interface is defined, any number of classes can implement it.
- Also, one class can implement any number of interfaces.

Interfaces

Here is a simplified general form of a traditional interface:

```
Access specifier interface  
name  
{  
    ret-type method-name1 (param-  
    list);  
    ret-type method-name2 (param-  
    list);  
    type var1 = value;  
    type var2 = value;  
    ::  
    ::  
}  
}
```

- Access specifier is either **public** or not used (**friendly**)
- methods are declared using only their return type and signature.
- They are, essentially, **abstract** methods and are implicitly **public**.
- Variables declared in an **interface** are not instance variables.
- Instead, they are implicitly **public**, **final**, and **static** and must be initialized.

Interfaces

Here is an example of an **interface** definition.

```
public interface Numbers
{
    int getNext(); // return next number in series
    void reset(); // restart
    void setStart(int x); // set starting value
}
```

Implementing Interfaces

The general form of a class that includes the **implements** clause looks like this:

```
Access Specifier class classname extends superclass
implements interface {
    // class-body
}
```

Implementing Interfaces

```
// Implement Numbers.  
class ByTwos implements Numbers  
{ int start;  
    int val;  
    public ByTwos() {  
        start = 0;  
        val = 0;  
    }  
    public int getNext() {  
        val += 2;  
        return val;  
    }  
    public void reset() {  
        val = start;  
    }  
    public void setStart(int x) {  
        start = x;  
        val = x;  
    } }
```

- Class **ByTwos** implements the **Numbers** interface
- Notice that the methods `getNext()`, `reset()`, and `setStart()` are declared using the `public` access specifier

Implementing Interfaces

```
public class Demo {  
  
    public static void main (String args[]) {  
  
        ByTwos ob = new ByTwos();  
  
        for (int i = 0; i < 5; i++) {  
  
            System.out.println("Next value is " + ob.getNext());  
  
            System.out.println("\n Resetting");  
  
            ob.reset();}  
  
        for (int i = 0; i < 5; i++)  
  
            System.out.println("Next value is " + ob.getNext());  
  
        System.out.println("\n Starting at 100");  
  
        ob.setStart(100);  
  
        for (int i = 0; i < 5; i++)  
  
            System.out.println("Next value is " + ob.getNext());  
    } }  
}
```

Implementing Interfaces

```
// Implement Numbers.  
class ByThrees implements  
Numbers  
{  int start;  
    int val;  
    public ByThrees() {  
        start = 0;  
        val = 0;  
    }  
    public int getNext() {  
        val += 3; return val;  
    }  
    public void reset() {  
        val = start;  
    }  
    public void setStart( int x)  
    { start = x;  
        val = x;  
    } }
```

- Class **ByThrees** provides another implementation of the **Numbers** interface
- Notice that the methods `getNext()`, `reset()`, and `setStart()` are declared using the `public` access specifier

Using interface reference

```
class Demo2
{
    public static void main (String args[])
    {
        ByTwos twoOb = new ByTwos();
        ByThrees threeOb = new ByThrees();
        Numbers ob;
        for(int i=0; i < 5; i++) {
            ob = twoOb;
            System.out.println("Next ByTwos value is " + ob.getNext());
            ob = threeOb;
            System.out.println("Next ByThrees value is " + ob.getNext());
        }
    }
}
```

General Consideration about interfaces

- Variables can be declared in an interface, but they are implicitly public, static, and final.
- To define a set of shared constants, create an interface that contains only these constants, without any methods.
- One interface can inherit another by use of the keyword extends. *The syntax is the same as for inheriting classes.*
- When a class implements an interface that inherits another interface, it must provide implementations for all methods required by the interface inheritance chain.

Default Methods

Java SE 8 New
Feature

- Prior to JDK 8, an interface **could not** define any implementation whatsoever.
- The release of JDK 8 changed this by adding a new capability to interface called the default method.
- A default method lets you define a **default implementation** for an interface method.
- You specify an interface default method by using the **default** keyword

Default Methods

```
interface InterfaceA {  
  
    public void saySomething();  
  
    default public void sayHi() {  
        System.out.println("Hi");  
    }  
}
```

Java SE 8 New Feature

```
public class MyClass  
implements InterfaceA  
{  
  
    public void  
saySomething() {  
  
    System.out.println("Hello  
World");  
    }  
}
```

Default Methods

- **Extending Interfaces That Contain Default Methods**

- Not mention the default method at all, which lets your extended interface inherit the default method.

```
interface Intf1 {  
    default void method() { doSomething(); }  
}  
interface Intf2 extends Intf1 {  
}
```



Java SE 8 New Feature

Default Methods

- Extending Interfaces That Contain Default Methods

Java SE 8 New
Feature

- Redefine the default method, which overrides it.

```
interface Intf1 {  
    default void method() { doSomething(); }  
}  
  
interface Intf2 extends Intf1 {  
    default void method() { doAnother(); }  
}
```

Default Methods

Java SE 8 New Feature

- Extending Interfaces That Contain Default Methods

- Re-declare the default method, which makes it abstract.

```
interface Intf1 {  
    default void method() { doSomething(); }  
}  
  
interface Intf2 extends Intf1 {  
    abstract void method();  
}
```

Use static Methods in an Interface

Java SE 8 New Feature

- JDK 8 added another new capability to interface:
 - The ability to define one or more **static methods**.
 - Like static methods in a class, a static method defined by an interface can be called independently of any **object**.
 - Thus, no implementation of the interface is necessary, and no instance of the interface is required in order to call a static method.

Use static Methods in an Interface

Java SE 8 New Feature

```
public interface MyIF {  
    // This is a "normal" interface method declaration.  
    // It does NOT define a default implementation.  
    int getUserId();  
    // This is a default method. Notice that it provides a  
    // default implementation.  
default int getAdminID()  
{  
    return 1;  
}  
    // This is a static interface method.  
static int getUniversalID()  
{  
    return 0;  
}
```

Private Interface Methods

Java SE 9 New Feature

- Beginning with JDK 9, an interface can include a private method.
- A private interface method can be called only by a default method or another private method defined by the same interface.
- Because a private interface method is specified **private**, it cannot be used by code outside the interface in which it is defined.
- This restriction includes subinterfaces because a private interface method is not inherited by a subinterface.

Private Interface Methods

Java SE 9 New Feature

- The key benefit of a private interface method is that it lets two or more default methods use a common piece of code, thus avoiding code duplication.
- Consider an example of a stack of integers, let us define an interface called ***IntStack*** that has two default methods called ***popNElements()*** and ***skipAndPopNElements()***.
- The first returns an array that contains the top N elements on the stack. The second skips a specified number of elements and then returns an array that contains the next N elements.

Private Interface Methods

- Both use a private method called **getElements()** to obtain an array of the specified number of elements from the stack.

```
//Another version of IntStack that has a private interface
//method that is used by two default methods.

interface IntStack {
    void push(int item); // store an item
    int pop(); // retrieve an item
    // A default method that returns an array that contains
    // the top n elements on the stack.
    default int[] popNElements(int n) {
        // Return the requested elements.
        return getElements(n);
    }
}
```

Private Interface Methods

```
// A default method that returns an array that contains
// the next n elements on the stack after skipping elements.
default int[] skipAndPopNElements(int skip, int n) {
    // Skip the specified number of elements.
    getElements(skip);
    // Return the requested elements.
    return getElements(n);
}
// A private method that returns an array containing
// the top n elements on the stack
private int[] getElements(int n) {
    int[] elements = new int[n];
    for (int i = 0; i < n; i++)
        elements[i] = pop();
    return elements;
}
```

Private Interface Methods

- Notice that both **popNElements()** and **skipAndPopNElements()** use the private **getElements()** method to obtain the array to return.
- This prevents both methods from having to duplicate the same code sequence.
- Keep in mind that because **getElements()** is private, it cannot be called by code outside **IntStack**. Thus, its use is limited to the default methods inside **IntStack**.

Private Interface Methods

Java SE 9 New Feature

- Also, because **getElements()** uses the **pop()** method to obtain stack elements, it will automatically call the implementation of **pop()** provided by the **IntStack** in its implementation class.
- Thus, **getElements()** will work for any stack class that implements **IntStack**.

Interfaces and Callbacks

- A common pattern in programming is the *callback* pattern.
- In this pattern, you specify the action that should occur whenever a particular event happens.
- Consider the following example:
- The javax.swing package contains a Timer class that is useful if you want to be notified whenever a time interval has elapsed.
- For example, if a part of your program contains a clock, you can ask to be notified every second so that you can update the clock face.

Interfaces and Callbacks

- When you construct a timer, you set the time interval and you tell it what it should do whenever the time interval has elapsed.
- How do you tell the timer what it should do?
- You pass an object of some class. The timer then calls one of the methods on that object.
- Of course, the timer needs to know what method to call. The timer requires that you specify an object of a class that implements the **ActionListener** interface of the **java.awt.event** package. Here is that interface:

Interfaces and Callbacks

```
public interface ActionListener  
{  
void actionPerformed(ActionEvent event);  
}
```

- The timer calls the actionPerformed method when the time interval has expired.

Interfaces and Callbacks

```
class TimePrinter implements  
ActionListener  
{  
    public void  
    actionPerformed(ActionEvent event)  
{  
        System.out.println("At the tone, the  
time is " + new Date());  
        Toolkit.getDefaultToolkit().beep();  
    }  
}
```

```
ActionListener listener = new  
TimePrinter();  
Timer t = new Timer(10000,  
listener);
```

Functional Interfaces

- Functional interfaces are new additions in java 8 which permit exactly one abstract method inside them.
- These interfaces are also called **Single Abstract Method interfaces (SAM Interfaces)**.
- These can be represented using Lambda expressions, Method reference and constructor references as well.
- Java 8 introduces an annotation i.e. **@FunctionalInterface** too, which can be used for compiler level errors when the interface you have annotated violates the contracts of Functional Interface.

Functional Interfaces

```
package functionalInterfaceExample;  
@FunctionalInterface  
public interface MyFirstFunctionalInterface {  
    public void firstWork();  
}
```

Functional Interfaces

```
package functionalInterfaceExample;  
  
@FunctionalInterface  
  
public interface MyFirstFunctionalInterface {  
  
    public void firstWork();  
  
    public void doSomeMoreWork();  
  
}
```

Compilation Error

Unexpected @FunctionalInterface annotation

@FunctionalInterface ^
MyFirstFunctionalInterface is not a
functional interface

multiple non-overriding abstract
methods found in interface
MyFirstFunctionalInterface

Functional Interfaces

```
package functionalInterfaceExample;

@FunctionalInterface

public interface MyFirstFunctionalInterface {

    public void firstWork();

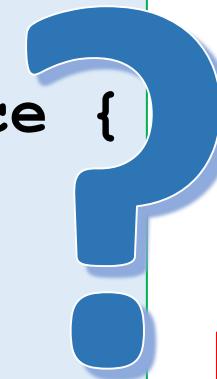
    @Override

    public String toString();

    @Override

    public boolean equals(Object obj);
```

Does not Compile



Compile

Functional Interfaces Examples

- Package `java.util.function` contains several functional interfaces.
- Throughout the table, T and R are generic type names that represent the type of the object on which the functional interface operates and the return type of a method, respectively.
- The Following Tables show six of the basic generic functional interfaces.

Functional Interfaces Examples

Interface	Description
<p><u>BinaryOperator<T></u></p> <p>“Represents an operation upon two operands of the same type, producing a result of the same type as the operands.”</p>	<p>Contains method apply that takes two T arguments, performs an operation on them (such as a calculation) and returns a value of type T.</p>
<p><u>Consumer<T></u></p> <p>“Represents an operation that accepts a single input argument and returns no result”</p>	<p>Contains method accept that takes a T argument and returns void. Performs a task with it's T argument, such as outputting the object, invoking a method of the object, etc.</p>
<p><u>Function<T,R></u></p> <p>“Represents a function that accepts one argument and produces a result.”</p>	<p>Contains method apply that takes a T argument and returns a value of type R. Calls a method on the T argument and returns that method's result.</p>

Functional Interfaces Examples

Interface	Description
<p><u>Predicate<T></u></p> <p>“Represents a predicate (boolean-valued function) of one argument.”</p>	<p>Contains method test that takes a T argument and returns a boolean. Tests whether the T argument satisfies a condition.</p>
<p><u>Supplier<T></u></p> <p>“Represents a supplier of results. There is no requirement that a new or distinct result be returned each time the supplier is invoked.”</p>	<p>Contains method get that takes no arguments and produces a value of type T. Often used to create a collection object in which a stream operation’s results are placed.</p>
<p><u>UnaryOperator<T></u></p> <p>“Represents an operation on a single operand that produces a result of the same type as its operand”</p>	<p>Contains method apply that takes an arguments of type T and returns a value of type T.</p>

Functional Interface Example

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.function.Predicate;

public class PredicateApp {
    private static List<String> getBeginWith(List<String> list, Predicate<String> valid)
    {
        List<String> selected = new ArrayList<>();
        list.forEach(player -> {
            if (valid.test(player)) {
                selected.add(player);
            }
        });
        return selected;
    }
}
```

Functional Interface Example

```
public static void main(String[] args) {  
  
    String[] players =  
  
    { "Rafael Nadal", "Novak Djokovic", "Stanislas Wawrinka", "David Ferrer", "Roger Federer",  
        "Andy Murray", "Tomas Berdych", "Juan Martin Del Potro", "Richard Gasquet", "John Isner"  
    };  
    List<String> playerList = Arrays.asList(players);  
  
    System.out.println(getBeginWith(playerList, (s) -> s.startsWith("R")));  
  
    System.out.println(getBeginWith(playerList, (s) -> s.contains("D")));  
  
    System.out.println(getBeginWith(playerList, (s) -> s.endsWith("er")));  
}  
}
```

Lab Exercise

- Develop an application to convert temperature from Centigrade to Fahrenheit using *Function<T,R>*
- Use the interfaces in `java.util.function` to build an application that defines the roots of the quadratic equation ($ax^2 + bx + c = 0$) and the roots could be computed by the following formula ($x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$)

A.	The History and Evolution of Java	2
B.	An Overview of Java	21
C.	Data Types, Variables and Arrays	64
D.	Operators, Control Statements and String Handling	98
E.	Packages and Interfaces	163
F.	Wrapper Classes, Enumeration, Autoboxing and Annotations	216
G.	Exception Handling	256
H.	Generics and Lambda Expressions	299
I.	Java Stream API	363
J.	Java Collections	430
K.	Java Platform Module System	522

Lesson 5

Wrapper Classes, Autoboxing, Enumerations, and Annotations

Wrapper Classes

- Each primitive data type has a corresponding wrapper class.

boolean	→	Boolean
byte	→	Byte
char	→	Character
short	→	Short
int	→	Integer
long	→	Long
float	→	Float
double	→	Double

Wrapper Classes cont'd

- There are three reasons that you might use a wrapper class rather than a primitive:
 1. As an argument of a method that expects an object.
 2. To use constants defined by the class,
 - such as **MIN_VALUE** and **MAX_VALUE**, that provide the upper and lower bounds of the data type.
 3. To use class methods for
 - converting values to and from other primitive types,
 - converting to and from strings,
 - converting between number systems (decimal, octal, hexadecimal, binary).

Wrapper Classes cont'd

- They have useful methods that perform some general operation, for example:

primitive xxxValue() → convert wrapper object to primitive

primitive parseXXX(String) → convert String to primitive

Wrapper valueOf(String) → convert String to Wrapper

```
Integer i2 = new Integer(42);
byte b = i2.byteValue();
double d = i2.doubleValue();
```

```
String s3 =
Integer.toHexString(254);
System.out.println("254 is " + s3);
```

Wrapper Classes cont'd

- They have special static representations, for example:

POSITIVE_INFINITY

NEGATIVE_INFINITY

NaN

Not a Number

In class Double
Float &

Autoboxing

Autoboxing

- Beginning with JDK 5, Java has included two important features: *autoboxing* and *auto-unboxing*.
- ***Autoboxing*** is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrapper whenever an object of that type is needed.
- ***Auto-unboxing*** is the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper when its value is needed.
- There is no need to call a method such as **`intValue()`** or **`doubleValue()`**.

Autoboxing

```
Integer intObject=100; //autobox an int  
int i = intObject; //auto-unbox
```

Autoboxing and Methods

- In addition to the simple case of assignments, autoboxing automatically occurs whenever a primitive type must be converted into an object; auto-unboxing takes place whenever an object must be converted into a primitive type.

Enumerations

Enumerations

- A *Java Enum* is a special Java type used to define *collections of constants*. More precisely, a Java enum type is a *special kind of Java class*.
- An enum can contain constants, methods etc. Java enums were added in Java 5.

Enumerations

- In its simplest form, an *enumeration* is a list of named constants that define a new data type and its legal values.
- Thus, an enumeration object can hold only a value that was declared in the list. Other values are not allowed.
- Enumerations are commonly used to define a set of values that represent a collection of items.
- In Java, an enumeration defines a class type. By making enumerations into classes, the capabilities of the enumeration are greatly expanded.

Enumerations

- For example, in Java, an enumeration can have constructors, methods, and instance variables.
- Because of their power and flexibility, enumerations are widely used throughout the Java API library.

Enumeration Example

```
public enum Level {  
    HIGH(3), // calls constructor with value 3  
    MEDIUM(2), // calls constructor with value 2  
    LOW(1) // calls constructor with value 1  
; // semicolon needed when fields / methods follow  
    private final int levelCode;  
    //enum constructor  
    Level(int levelCode) {  
        this.levelCode = levelCode;  
    }  
    // enum Methods  
    public int getLevelCode() {  
        return this.levelCode;  
    }  
}
```

Enumeration Fundamentals

- An enumeration is created using the **enum** keyword. For example, here is a simple enumeration that lists various week days:

```
// An enumeration of week days.  
enum week {Saturday, Sunday, Monday, Tuesday, Wednesday,  
Thursday, Friday}
```



Enumeration constants are implicitly declared as public static final members of week

Enumeration Fundamentals

- Once you have defined an enumeration, you can create a variable of that type.
- even though enumerations define a class type, you do not instantiate an **enum** using **new**.
- Instead, you declare and use an enumeration variable in much the same way as you do one of the primitive types. For example, this declares **w** as a variable of enumeration type **week**:

```
week w;
```

Enumeration Fundamentals

- Although you can't inherit a superclass when declaring an **enum**, all enumerations automatically inherit one: **java.lang.Enum**.
- This class defines several methods that are available for use by all enumerations.

final int ordinal():

Returns the ordinal of this enumeration constant (its position in its enum declaration, where the initial constant is assigned an ordinal of zero).

final int compareTo(enum-type e):

Compares this enum with the specified object for order.

Another Enumeration Example

```
enum Week {  
    Saturday, Sunday, Monday, Tuesday, Wednesday, Thursday, Friday  
}  
  
class EnumDemo {  
    public static void main(String args[])  
    {  
        Week ap;  
        ap = Week.Sunday;  
        // Output an enum value.  
        System.out.println("Value of ap: " + ap);  
        System.out.println();  
        ap = Week.Wednesday;  
        // Compare two enum values.  
        if(ap == Week.Wednesday)  
            System.out.println("Current week day is Wednesday.\n");
```

Enumeration Example

```
// Use an enum to control a switch statement.  
  
switch(ap) {  
    case Saturday:  
        System.out.println("Today is Saturday."); break;  
    case Sunday:  
        System.out.println("Today is Sunday."); break;  
    case Monday:  
        System.out.println("Today is Monday."); break;  
    case Tuesday:  
        System.out.println("Today is Tuesday."); break;  
    case Friday:  
        System.out.println("Today is Friday."); break;  
}
```

Annotations

Annotations

- Java provides a feature that enables you to embed supplemental information into a source file.
- This information, that is called an *annotation*; does not change the actions of a program. Thus, an annotation leaves the semantics of a program unchanged.
- The term *metadata* is also used to refer to this feature, but the term *annotation* is the most descriptive and more commonly used.

The Built-In Annotations

- Java defines many built-in annotations. Most are specialized, but nine are general purpose.
- Of these, four are imported from `java.lang.annotation`:
`@Retention`, `@Documented`, `@Target`, and `@Inherited`.
- Five—`@Override`, `@Deprecated`, `@FunctionalInterface`, `@SafeVarargs`, and `@SuppressWarnings`—are included in `java.lang`.

The Built-In Annotations

java.lang

Annotation	Purpose
java.lang.Deprecated	A program element annotated @Deprecated is one that programmers are discouraged from using
java.lang.FunctionalInterface	An informative annotation type used to indicate that an interface type declaration is intended to be a <i>functional interface</i>
java.lang.Override	Indicates that a method declaration is intended to override a method declaration in a super type.
java.lang.SafeVarargs	A programmer assertion that the body of the annotated method or constructor does not perform potentially unsafe operations on its varargs parameter.
java.lang.SuppressWarnings	Indicates that the named compiler warnings should be suppressed in the annotated element

The Built-In Annotations

java.lang.annotation

Annotation	Purpose
java.lang.annotation.Documented	@Documented is a meta-annotation. You apply @Documented when defining an annotation, to ensure that classes using your annotation show this in their generated JavaDoc.
java.lang.annotation.Inherited	Indicates that an annotation type is automatically inherited.
java.lang.annotation.Native	Indicates that a field defining a constant value may be referenced from native code.
java.lang.annotation.Repeatable	The annotation type java.lang.annotation.Repeatable is used to indicate that the annotation type whose declaration it (meta-)annotates is <i>repeatable</i> .
java.lang.annotation.Retention	Indicates how long annotations with the annotated type are to be retained.
java.lang.annotation.Target	Indicates the contexts in which an annotation type is applicable.

Annotations Basics

- An annotation is created through a mechanism based on the **interface**.
- Let's begin with an example. Here is the declaration for an annotation called **MyAnno**:

Annotations Basics

```
import java.lang.annotation.*;  
// An annotation type declaration.  
@Retention(RetentionPolicy.RUNTIME)  
@interface MyAnno {  
    String str();  
    int val();  
}
```

Annotations Basics

- First, notice the @ that precedes the keyword **interface**. This tells the compiler that an annotation type is being declared.
- Next, notice the two members **str()** and **val()**. All annotations consist solely of method declarations. However, you don't provide bodies for these methods. Instead, Java implements these methods.
- An annotation cannot include an **extends** clause. However, all annotation types automatically extend the **Annotation** interface.
- The **Annotation interface** is a super interface of all annotations. It is declared within the **java.lang.annotation** package.

Annotations Basics

- Once you have declared an annotation, you can use it to annotate something. Prior to JDK 8, annotations could be used only on declarations. JDK 8 added the ability to annotate type use.
- When you apply an annotation, you give values to its members. For example, here is an example of **MyAnno** being applied to a method declaration:

```
// Annotate a method.
```

```
@MyAnno(str = "Annotation Example", val = 100)
```

```
public static void myMeth() {
```

```
    Meta ob = new Meta(); { //...
```

Annotations Basics

- The name of the annotation, preceded by an @, is followed by a parenthesized list of member initializations.
- To give a member a value, that member's name is assigned a value.
- Therefore, in the example, the string "Annotation Example" is assigned to the **str** member of **MyAnno**.
- When an annotation member is given a value, only its name is used. Thus, annotation members look like fields in this context.

Annotations Basics - Retention Policy

- Before exploring annotations further, it is necessary to discuss *annotation retention policies*.
- A retention policy determines at what point an annotation is discarded. Java defines three such policies, which are encapsulated within the **java.lang.annotation.RetentionPolicy** enumeration.
- They are **SOURCE**, **CLASS**, and **RUNTIME**.

Annotations Basics - Retention Policy

- An annotation with a retention policy of **SOURCE** is retained only in the source file and is discarded during compilation.
- An annotation with a retention policy of **CLASS** is stored in the **.class** file during compilation. However, it is not available through the JVM during run time.
- An annotation with a retention policy of **RUNTIME** is stored in the **.class** file during compilation and is available through the JVM during run time. Thus, **RUNTIME** retention offers the greatest annotation persistence.

Annotations Basics - Retention Policy

- A retention policy is specified for an annotation by using one of Java's built-in annotations: **@Retention**. Its general form is shown here:

`@Retention(retention-policy)`

Example:

`@Retention(RetentionPolicy.RUNTIME)`

Obtaining Annotations at Run Time by Use of Reflection

- Although annotations are designed mostly for use by other development or deployment tools, if they specify a retention policy of **RUNTIME**, then they can be queried at run time by any Java program through the use of *reflection*.
- Reflection is the feature that enables information about a class to be obtained at run time.
- The reflection API is contained in the **java.lang.reflect** package.
- The first step to using reflection is to obtain a **Class** object that represents the class whose annotations you want to obtain.
- **Class** is one of Java's built-in classes and is defined in **java.lang**.

Obtaining Annotations at Run Time by Use of Reflection

- There are various ways to obtain a **Class** object. One of the easiest is to call **getClass()**, which is a method defined by **Object**. Its general form is shown here:

final Class<?> getClass()

It returns the **Class** object that represents the invoking object.

Notice the **<?>** that follows **Class** in the declaration of **getClass()** just shown. This is related to Java's generics feature. **getClass()** and several other reflection-related methods discussed in this section make use of generics.

Obtaining Annotations at Run Time by Use of Reflection

- After you have obtained a **Class** object, you can use its methods to obtain information about the various items declared by the class, including its annotations.
- If you want to obtain the annotations associated with a specific item declared within a class, you must first obtain an object that represents that item.
- For example, **Class** supplies the **getMethod()**, **getField()**, and **getConstructor()** methods, which obtain information about a method, field, and constructor, respectively.

Obtaining Annotations at Run Time by Use of Reflection

- These methods return objects of type **Method**, **Field**, and **Constructor**.
- let's work through an example that obtains the annotations associated with a method.
- To do this, you first obtain a **Class** object that represents the class, and then call **getMethod()** on that **Class** object, specifying the name of the method. **getMethod()** has this general form:

Method getMethod(String methName, Class<?> ... paramTypes)

Obtaining Annotations at Run Time by Use of Reflection

- From a **Class**, **Method**, **Field**, or **Constructor** object, you can obtain a specific annotation associated with that object by calling **getAnnotation()**. Its general form is shown here:

<A extends Annotation> getAnnotation(Class<A> annoType)

- Here, *annoType* is a **Class** object that represents the annotation in which we are interested.

Let us have a look at a complete example

Obtaining Annotations at Run Time by Use of Reflection - Example

```
import java.lang.reflect.*;
class Meta {
    // Annotate a method.
    @MyAnno(str = "Annotation Example", val = 100)
    public static void myMeth() {
        Meta ob = new Meta();
        // Obtain the annotation for this method and display the
        // values of the members.
        try {
            // First, get a Class object that represents this class.
            Class<?> c = ob.getClass();
            // Now, get a Method object that represents this method.
            Method m = c.getMethod("myMeth");
```

Obtaining Annotations at Run Time by Use of Reflection - Example

```
// Next, get the annotation for this class.  
MyAnno anno = m.getAnnotation(MyAnno.class);  
// Finally, display the values.  
System.out.println(anno.str() + " " + anno.val());  
} catch (NoSuchMethodException exc) {  
System.out.println("Method Not Found.");  
}  
}  
  
public static void main(String args[]) {  
myMeth();  
}  
}
```

Lab Exercise

- Create a custom annotation named Author and allow it to be used on the class, method, constructor, and member level.
- Create a class annotated with @Author
- Using reflection to extract information regarding the annotated members

A.	The History and Evolution of Java	2
B.	An Overview of Java	21
C.	Data Types, Variables and Arrays	64
D.	Operators, Control Statements and String Handling	98
E.	Packages and Interfaces	163
F.	Wrapper Classes, Enumeration, Autoboxing and Annotations	216
G.	Exception Handling	256
H.	Generics and Lambda Expressions	299
I.	Java Stream API	363
J.	Java Collections	430
K.	Java Platform Module System	522

Lesson 6

Exception handling

Exceptions

Dealing with Errors

- Encountering errors is unpleasant. If a user loses all the work he or she did during a program session because of a programming mistake or some external circumstance, that user may forever turn away from your program.
- At the very least, you must:
 - Notify the user of an error;
 - Save all work; and
 - Allow users to gracefully exit the program.

Dealing with Errors

- For exceptional situations, such as bad input data with the potential to ruin the program, Java uses a form of error trapping called, naturally enough, ***exception handling***.
- Suppose an error occurs while a Java program is running.
 - The error might be caused by a file containing wrong information.
 - A bad network connection, or
 - use of an **invalid array index** or an attempt to use an object reference that hasn't yet been assigned to an object.

Dealing with Errors

- Users expect that programs will act sensibly when errors happen. If an operation cannot be completed because of an error, the program ought to either:
 - Return to a safe state and enable the user to execute other commands; or
 - Allow the user to save all work and terminate the program gracefully.
- ***The mission of exception handling is to transfer control from where the error occurred to an error handler that can deal with the situation.***

Dealing with Errors

- To handle exceptional situations in your program, you must take into account the errors and problems that may occur.
- ***What sorts of problems do you need to consider?***
 - *User input errors.*
 - *Device errors.*
 - *Physical limitations (Disks can fill up; you can run out of available memory)*
 - *Code errors*

Dealing with Errors

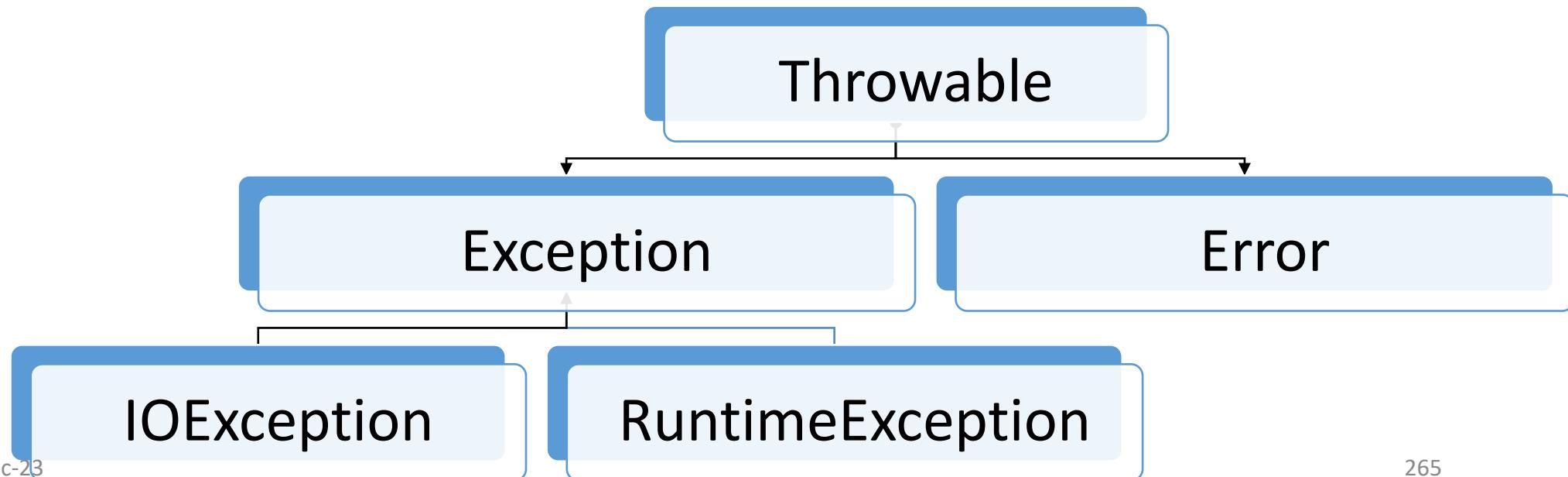
- The traditional reaction to an error in a method is to return a special error code that the calling method analyzes.
- For example, methods that read information back from files often return a -1 end-of-file value marker rather than standard character.
- Unfortunately, it is not always possible to return an error code. There may be no obvious way of distinguishing valid and invalid data.
- A method returning an integer cannot simply return -1 to denote the error; the value -1 might be a perfectly valid result.

Dealing with Errors

- ***Java allows every method an alternative exit path if it is unable to complete its task in the normal way.***
- In this situation, the method does not return a value. Instead, it ***throws*** an object that encapsulates the error information.
- Note that the method exits immediately; it does not return its normal (or any) value.
- Moreover, execution does not resume at the code that called the method; instead, the exception-handling mechanism begins its search for an ***exception handler*** that can deal with this particular error condition.

The Classification of Exceptions

- In the Java programming language, an exception object is always an instance of a class derived from ***Throwable***.
- As you will soon see, you can create your own exception classes if the ones built into Java do not suit your needs.



The Classification of Exceptions

- Notice that all exceptions descend from Throwable, but the hierarchy immediately splits into two branches: Error and Exception.
- The Error hierarchy describes internal errors and resource exhaustion situations inside the Java runtime system. ***You should not throw an object of this type.***
- The Exception hierarchy also splits into two branches: exceptions that derive from RuntimeException and those that do not.

The Classification of Exceptions

- The general rule is this: A *RuntimeException* happens because you made a programming error. Any other exception occurs because a bad thing, such as an I/O error, happened to your otherwise good program.
- Exceptions that inherit from RuntimeException include such problems as:
 - *A bad Cast*
 - *An out-of-bound array access*
 - *A null pointer access*

The Classification of Exceptions

- Exceptions that do not inherit from *RuntimeException* include:
 - *Trying to read past the end of a file*
 - *Trying to open a file that does not exist*
 - *Trying to find a Class object for a string that does not denote an existing class*
- The rule “*If it is a RuntimeException, it was your fault*” works pretty well. You could have avoided that *ArrayIndexOutOfBoundsException* by testing the array index against the array bounds.

The Classification of Exceptions

- The *NullPointerException* would not have happened had you checked whether the variable was *null* before using it.
- The Java Language Specification calls any exception that derives from the class *Error* or the class *RuntimeException* an *unchecked* exception.
- All other exceptions are called *checked* exceptions because the compiler checks that you provide exception handlers for all checked exceptions.

Declaring Checked Exceptions

- A Java method can throw an exception if it encounters a situation it cannot handle.
- The idea is simple: A method will not only tell the Java compiler what values it can return, *it is also going to tell the compiler what can go wrong.*
- The place in which you advertise that your method can throw an exception is the header of the method; the header changes to reflect the checked exceptions the method can throw.

Declaring Checked Exceptions

- For example, here is the declaration of one of the constructors of the `FileInputStream` class from the standard library.

public FileInputStream(String name) throws FileNotFoundException

- When you write your own methods, you don't have to advertise every possible throwable object that your method might actually throw.

Declaring Checked Exceptions

- keep in mind that an exception is thrown in any of the following four situations:

If either of the first two scenarios occurs, you must tell the programmers who will use your method about the possibility of an exception. Why? If no handler catches the exception, the current thread of execution terminates.

- 3) You make a programming error, such as $a[-1] = 0$ that gives rise to an unchecked exception (in this case, an *ArrayIndexOutOfBoundsException*).
- 4) An internal error occurs in the virtual machine or runtime library.

Declaring Checked Exceptions

- As with Java methods that are part of the supplied classes, you declare that your method may throw an exception with an *exception specification* in the method header.

```
class MyAnimation
{
...
public Image loadImage(String s)
throws IOException
{
...
}
```

```
class MyAnimation
{
...
public Image loadImage(String s) throws
FileNotFoundException, EOFException
{
...
}
```

Declaring Checked Exceptions

- In summary, a method must declare all the *checked* exceptions that it might throw.
- Unchecked exceptions are either beyond your control (*Error*) or result from conditions that you should not have allowed in the first place (*RuntimeException*).
- *If you override a method from a superclass, the checked exceptions that the subclass method declares cannot be more general than those of the superclass method. (It is OK to throw more specific exceptions, or not to throw any exceptions in the subclass method.)*

How to Throw an Exception

- You have a method, *readData*, that is reading in a file whose header promised

Content-length: 1024

- but you got an end of file after 733 characters. You may decide this situation is so abnormal that you want to throw an *EOFException* exception.

throw new EOFException();

or, if you prefer,

EOFException e = new EOFException();

throw e;

How to Throw an Exception

- Here is how it all fits together:

```
String readData(Scanner in) throws EOFException
{
    . . .
    while (...)
    {
        if (!in.hasNext()) // EOF encountered
        {
            if (n < len)
                throw new EOFException();
        }
        . . .
    }
    return s;
}
```

How to Throw an Exception

- As you can see, throwing an exception is easy if one of the existing exception classes works for you. In this case:
 1. Find an appropriate exception class.
 2. Make an object of that class.
 3. Throw it.
- Once a method throws an exception, it does not return to its caller.
- This means you do not have to worry about setting up a default return value or an error code.

Creating Exception Classes

- Your code may run into a problem which is not satisfactorily described by any of the standard exception classes.
- In this case, it is easy enough to create your own exception class.
- Just derive it from `Exception`, or from a child class of `Exception` such as `IOException`.
- It is customary to give both a default constructor and a constructor that contains a detailed message.
- Note that The `toString` method of the `Throwable` superclass returns a string containing that detailed message

Creating Exception Classes

```
import java.io.IOException;

public class FileFormatException extends IOException {

    public FileFormatException() {
        // TODO Auto-generated constructor stub
    }

    public FileFormatException(String arg0) {
        super(arg0);
        // TODO Auto-generated constructor stub
    }
}
```

Catching Exceptions

- You now know how to throw an exception.
- It is pretty easy: You throw it and you forget it.
- If an exception occurs that is not caught anywhere, the program will terminate and print a message to the console, giving the type of the exception and a stack trace.
- GUI programs catch exceptions, print stack trace messages, and then go back to the user interface processing loop.

Catching Exceptions

- To catch an exception, set up a try/catch block. The simplest form of the try block is as follows:

```
try
{
    code
    more code
    more code
}
catch (ExceptionType e)
{
    handler for this type
}
```

If any code inside the *try* block throws an exception of the class specified in the *catch* clause, then

- The program skips the remainder of the code in the try block.
- The program executes the handler code inside the catch clause

Catching Exceptions

- If none of the code inside the ***try*** block throws an exception, then the program skips the ***catch*** clause.
- Here is an example for a typical code for reading data

Catching Exceptions

```
public void read(String filename) {  
    try {  
        InputStream in = new FileInputStream(filename);  
        int b;  
        while ((b = in.read()) != -1) {  
            // process input  
        }  
    } catch (IOException exception) {  
        exception.printStackTrace();  
    }  
}
```

May throw IOException

We can remove the catch clause and modify the method header to throw the exception thus delegating to the caller the exception handling process

Catching Exceptions

- Which way is better?
 - To handle the exception
 - To propagate to the caller
- *As a general rule, you should catch those exceptions that you know how to handle and propagate those that you do not know how to handle.*

Catching Multiple Exceptions

- You can catch multiple exception types in a ***try*** block and handle each type differently. Use a separate ***catch*** clause for each type as in the following example:

Catching Multiple Exceptions

```
try
{
    code that might throw exceptions
}
catch (FileNotFoundException e)
{
    emergency action for missing files
}
catch (UnknownHostException e)
{
    emergency action for unknown hosts
}
catch (IOException e)
{
    emergency action for all other I/O problems
}
```

The exception object may contain information about the nature of the exception.

To find out more about the object, try **e.getMessage()**

to get the detailed error message or to get the actual type of the exception object.

e.getClass().getName()

Catching Multiple Exceptions

- As of Java SE7, you can catch multiple exception types in the same catch clause.
- For example, suppose that the action for missing files and unknown hosts is the same. Then you can combine the catch clauses:

```
try
{
    code that might throw exceptions
}
catch (FileNotFoundException | UnknownHostException e)
{
    emergency action for missing files and unknown hosts
}
catch (IOException e)
{
    emergency action for all other I/O problems
}
```

This feature is only needed when catching exception types that are not subclasses of one another.

Rethrowing and Chaining Exceptions

- You can throw an exception in a catch clause. Typically, you do this when you want to change the exception type.
- If you build a subsystem that other programmers use, it makes a lot of sense to use an exception type that indicates a failure of the subsystem.
- An example of such an exception type is the `ServletException`.
- The code that executes a servlet may not want to know in minute detail what went wrong, but it wants to know that the servlet was at fault.

Rethrowing and Chaining Exceptions

```
try
{
    access the database
}
catch (SQLException e)
{
    throw new ServletException("database error: " + e.getMessage());
}
```

- Here, the ServletException is constructed with the message text of the exception.

Rethrowing and Chaining Exceptions

- However, it is a better idea to set the original exception as the “cause” of the new exception:

```
try
{
    access the database
}
catch (SQLException e)
{
    Throwable se = new ServletException("database error");
    se.initCause(e);
    throw se;
}
```

- This wrapping technique is highly recommended. It allows you to throw high level exceptions in subsystems without losing the details of the original failure.

The *finally* Clause

- When your code throws an exception, it stops processing the remaining code in your method and exits the method.
- This is a problem if the method has acquired some local resource, which only this method knows about, and that resource must be cleaned up.
- One solution is to catch and re-throw all exceptions. But this solution is tedious because you need to clean up the resource allocation in two places—in the normal code and in the exception code.

The *finally* Clause

- Java has a better solution: the *finally* clause.
- The code in the finally clause executes whether or not an exception was caught.
- In the following example, the program will dispose of the *FileInputStream* under all circumstances:

The *finally* Clause

```
InputStream in = new FileInputStream(filename);
try {
// 1
code that might throw exceptions
// 2
} catch (IOException e) {
// 3
show error message
// 4
} finally {
// 5
in.close();
}
// 6
```

The *finally* Clause



CAUTION: A finally clause can yield unexpected results when it contains return statements.

Suppose you exit the middle of a try block with a return statement.

Before the method returns, the finally block is executed.

If the finally block also contains a return statement, then it masks the original return value.

The Try-with-Resources Statement

- Starting Java SE 7 the ***Try-with-Resources*** statement have been introduced.
- In its simplest variant, the try-with-resources statement has the form

```
try (Resource res = . . .)
```

```
{
```

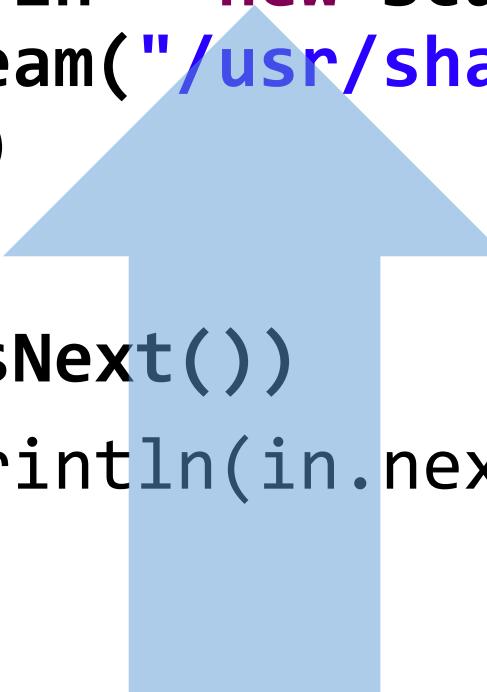
```
work with res
```

```
}
```

- When the try block exits, then `res.close()` is called automatically. Here is a typical example—reading all words of a file:

The Try-with-Resources Statement

```
try (Scanner in = new Scanner(new  
FileInputStream("/usr/share/dict/words  
")), "UTF-8")  
{  
    while (in.hasNext())  
        System.out.println(in.next());  
}
```



When the block exits normally, or when there was an exception, the `in.close()` method is called, exactly as if you had used a `finally` block.

The resource must belong to a class that implements the *AutoCloseable* interface.

The Try-with-Resources Statement

- You can specify multiple resources. For example:

```
try (Scanner in = new Scanner(new  
FileInputStream("/usr/share/dict/words"), "UTF-8");  
PrintWriter out = new PrintWriter("out.txt"))  
{  
    while (in.hasNext())  
        out.println(in.next().toUpperCase());  
}
```

Lab Exercise

- Create your own exception class and write down two other classes, the first will contain three methods throwing your newly created exception class and the second class will be calling the methods that throws exception using the try-catch-finally block.

A.	The History and Evolution of Java	2
B.	An Overview of Java	21
C.	Data Types, Variables and Arrays	64
D.	Operators, Control Statements and String Handling	98
E.	Packages and Interfaces	163
F.	Wrapper Classes, Enumeration, Autoboxing and Annotations	216
G.	Exception Handling	256
H.	Generics and Lambda Expressions	299
I.	Java Stream API	363
J.	Java Collections	430
K.	Java Platform Module System	522

Lesson 7

Generic Programming

Generic Programming

Introduction

Why generic Programming

- ***Generic programming*** means writing code that can be reused for objects of many different types.
- Generic classes are desirable because they let you write code that is safer and easier to read than code littered with Object variables and casts.
- Generics are particularly useful for collection classes, such as the universal ArrayList.
- Generic classes are—at least on the surface—similar to templates in C++. In C++, as in Java, templates were first added to the language to support strongly typed collections.

Why generic Programming

- It is important to understand that Java has always given you the ability to create generalized classes, interfaces, and methods by operating through references of type **Object**.
- Because **Object** is the superclass of all other classes, an **Object** reference can refer to any type object.
- In pre-generics code, generalized classes, interfaces, and methods used **Object** references to operate on various types of objects.
- The problem was that they could not do so with type safety.

Why generic Programming

- Generics added the type safety that was lacking.
- They also streamlined the process, because it is no longer necessary to explicitly employ casts to translate between **Object** and the type of data that is actually being operated upon.

Why generic Programming

- In other words the term generics means parametrized types.
- *Parameterized types are important because they enable you to create classes, interfaces, and methods in which the type of data upon which they operate is specified as a parameter.*
- Using generics, it is possible to create a single class, for example, that automatically works with different types of data.
- A class, interface, or method that operates on a parameterized type is called *generic*, as in *generic class* or *generic method*.

The Advantages of Type Parameters

- Before generic classes were added to Java, generic programming was achieved with *inheritance*.
- The ArrayList class simply maintained an array of Object references:

```
public class ArrayList // before generic classes
{
    private Object[] elementData;
    ...
    public Object get(int i) { . . . }
    public void add(Object o) { . . . }
}
```

The Advantages of Type Parameters

- This approach has two problems. A cast is necessary whenever you retrieve a value:

```
ArrayList files = new ArrayList();
```

```
    . . .
```

```
String filename = (String) files.get(0);
```

- Moreover, there is no error checking. You can add values of any class:

```
files.add(new File("." . .));
```

The Advantages of Type Parameters

- The call in the previous slide compiles and runs without error.
- Elsewhere, casting the result of *get* to a String will cause an error.
- Generics offer a better solution: ***type parameters***. The *ArrayList* class now has a type parameter that indicates the element type:

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

- This makes your code easier to read. You can tell right away that this particular array list contains String objects.

The Advantages of Type Parameters

- The compiler can make good use of the type information too. No cast is required for calling ***get***. The compiler knows that the return type is String, not Object:

String filename = files.get(0);

- The compiler also knows that the ***add*** method of an ***ArrayList<String>*** has a parameter of type ***String***.
- That is a lot safer than having an Object parameter. Now the compiler can check that you don't insert objects of the wrong type. For example, the statement

files.add(new File("...")); // can only add String objects to an ArrayList<String>

will not compile. A compiler error is much better than a class cast exception at runtime.

Who Wants to Be a Generic Programmer?

- It is easy to use a generic class such as *ArrayList*.
- Most Java programmers will simply use types such as *ArrayList<String>* as if they had been built into the language, just like *String[]* arrays.
- However, it is not so easy to implement a generic class.
- The programmers who use your code will want to plug in all sorts of classes for your type parameters.
- They will expect everything to work without tedious restrictions and confusing error messages.

Who Wants to Be a Generic Programmer?

- Your job as a generic programmer, therefore, is to anticipate all the potential future uses of your class.
- Generic programming falls into three skill levels.
 1. At a basic level, you just use generic classes
 2. you need to learn enough about Java generics to solve problems systematically rather than through random fixing.
 3. Finally, of course, you may want to implement your own generic classes and methods.

Simple Generic Example

```
public class Pair<T> {  
    private T first;  
    private T second;  
    public Pair() {  
        first = null;  
        second = null;  
    }  
    public Pair(T first, T second) {  
        this.first = first;  
        this.second = second;  
    }  
    public T getFirst() {  
        return first;  
    }
```



```
    public T getSecond() {  
        return second;  
    }  
    public void setFirst(T newValue)  
    {  
        first = newValue;  
    }  
    public void setSecond(T newValue)  
    {  
        second = newValue;  
    }  
}
```

Simple Generic Example

```
class ArrayAlg {  
  
    /**  
     * Gets the minimum and maximum of an array of strings.  
     * @param a an array of strings  
     * @return a pair with the min and max value, or null if a is null or empty  
     */  
  
    public static Pair<String> minmax(String[] a) {  
        if (a == null || a.length == 0)  
            return null;  
  
        String min = a[0];  
        String max = a[0];  
  
        for (int i = 1; i < a.length; i++) {  
            if (min.compareTo(a[i]) > 0) min = a[i];  
            if (max.compareTo(a[i]) < 0) max = a[i];  
        }  
        return new Pair<>(min, max);  
    } }
```

Simple Generic Example

```
public class PairTest {  
    public static void main(String[] args) {  
        String[] words = { "Mary", "had", "a",  
                           "little", "lamb" };  
        Pair<String> mm = ArrayAlg.minmax(words);  
        System.out.println("min = " + mm.getFirst());  
        System.out.println("max = " + mm.getSecond());  
    }  
}
```

Generics: parameter type bounds

- When defining a generic class/method, a type bound can be stated on any type parameter.
- A type bound can be any **reference type**, i.e. a class type that is used to further describe a type parameter. It **restricts** the set of types that can be used as type arguments and gives access to the non-static methods of the type it mentions.
 - A type parameter can be unbounded. In this case any reference type can be used as type argument to replace the unbounded type parameter in an instantiation of a generic type.
 - Alternatively we can have one or several bounds. In this case the type argument that replaces the bounded type parameter in an instantiation of a generic type must be a subtype of all bounds.

Generics: parameter type bounds

- The syntax for specification of type parameter bounds is:

```
<TypeParameter extends AClass & AnInterface1 & ... & AnInterfaceN>
```

i.e. a list of bounds consists of one class and/or several interfaces.

- The reason for imposing a type bound on a type parameter is that the code used in the **implementation code** of the generic class is **assuming** that the type used is of a certain type or any of its subtypes, or that it implements a certain interface.
- This can lead to rather complex class declarations such as:

```
class Pair<A extends Comparable<A> & Cloneable ,  
          B extends Comparable<B> & Cloneable >  
    implements Comparable<Pair<A,B>>, Cloneable { ... }
```

- A pair class taking two parameters, where each parameter is of a certain type that implements **Comparable** with itself and implements **Cloneable**.
- The class itself implements **Comparable** with itself and implements **Cloneable**, i.e. instances of this class are **Comparable** with each other and are **Cloneable** and the same is assumed of both parts of the **Pair**.

Generics: parameter types, interesting cases

```
class PairUtil {
    public static <A extends Number, B extends Number> double add(Pair<A, B> p) {
        return p.getFirst().doubleValue() + p.getSecond().doubleValue();
    }

    public static <A, B> Pair<B, A> swap(Pair<A, B> p) {
        A first = p.getFirst();
        B second = p.getSecond();
        return new Pair<B, A>(second, first);
    }
}
```

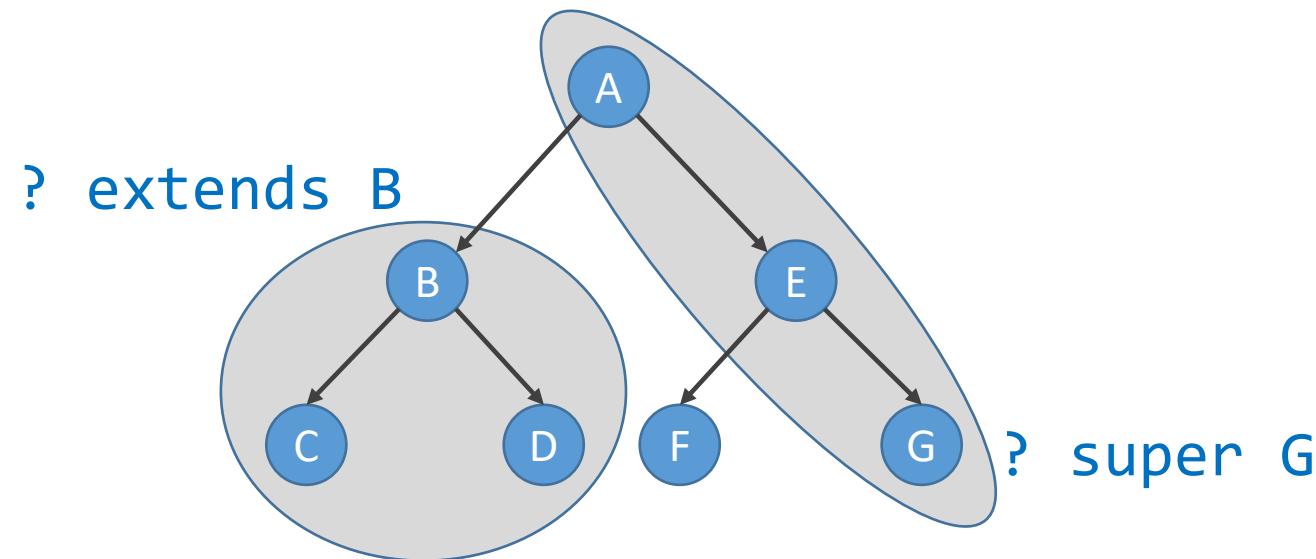
```
public class Pair <X, Y> {
    private final X a;
    private final Y b;

    public Pair(X a, Y b) {
        this.a = a;
        this.b = b;
    }
    public X getFirst() {
        return a;
    }
    public Y getSecond() {
        return b;
    }
}
```

- In the `add()` method:
 - The method is generic over two type parameters `A` and `B`, which must be subclasses of `Number`, which allows to use `doubleValue()`.
 - The argument to the method is a `Pair`; the type arguments to that `Pair` are constrained by the type parameter bounds to `add()`.
- In the `swap()` method:
 - The type parameters are used to define the return type, as well as the argument.
 - Local variables in the method are declared in terms of the type parameters.
 - The type parameters are used as type arguments in the constructor call.

Java Generics: wildcards

- A wildcard is a syntactic construct that is used to denote a family of types in a generic class/method **instantiation**. There are three different kinds of wildcards:
 - " ? " : the unbounded wildcard. It stands for the family of all types.
 - " ? extends SuperType " : a wildcard with an upper bound. It stands for the family of all types that are **SuperType** itself or subtypes of **SuperType**.
 - " ? super SubType " - a wildcard with a lower bound. It stands for the family of all types that are **SubType** itself or supertypes of **SubType**.



Java Generics: wildcards

```
public void printCollection( Collection<?> c ) {
    for (Object o : c){
        System.out.println(o);
    }
}
public static <T> void copy( List<? super T> dest, List<? extends T> src) {
    for (int i=0; i<src.size(); i++)
        dest.set(i,src.get(i));
}
```

- The method `printCollection()` receives a parameter that is a `Collection` of elements of any type. It can do so because it does not apply any type-specific operation on the `Collection` it receives as a parameter.
- The method `copy()` receives as parameters two lists, where the type of elements in the source `List` must be a subtype of type of elements in the destination `List`. Failure to impose such a restriction may allow to attempt to copy the elements of a list into a list of elements of an unrelated type, leading to an illegal type cast. It must do so because it uses `get()` and `set()`, which are bound to the type of values stored in the `List`.

Lab Exercise-1

- Create a base class named Shape that contains one abstract method draw().
- Create two concrete classes (Rectangle and Circle) that extend Shape
- Create a test class that defines a method that accepts a list that contains **only** child classes of shape
- Test your method by creating two ArrayList of Rectangle and shapes and pass them to the generic method

Lab Exercise-2

- Create a generic class that could be used to represent complex numbers
- Create some generic methods that represent basic arithmetic operation on complex(addition, subtraction, etc...)

Lesson 8

Lambda Expressions

Lambda Expressions

- Lambda expressions could be considered as the most exciting change to the Java language in many years.
- You will see how to use lambda expressions for defining blocks of code with a concise syntax, and how to write code that consumes lambda expressions.

Why Lambdas?

- *A lambda expression is a block of code that you can pass around so it can be executed later, once or multiple times.*
- In the previous slides we gave a brief introduction regarding “*Interfaces and Callbacks*”.
- you saw how to do work in timed intervals. Put the work into the actionPerformed method of an ActionListener.
- You then submit the instance to a Timer object.
- The key point is that the actionPerformed method contains code that you want to execute later.

Why Lambdas?

- A block of code was passed to someone—a timer, or a sort method. That code block was called at some later time.
- Up to now, giving someone a block of code hasn't been easy in Java.
- You couldn't just pass code blocks around. Java is an object-oriented language, so you had to construct an object belonging to a class that has a method with the desired code.
- In other languages, it is possible to work with blocks of code directly.

Why Lambdas?

- For some time now, the question was not whether to augment Java for functional programming, but how to do it. It took several years of experimentation before a design emerged that is a good fit for Java.
- In the next slides, we will see how we can work with blocks of code in Java SE 8.
- The logician Alonzo Church wanted to formalize what it means for a mathematical function to be effectively computable.
- Ever since, an expression with parameter variables has been called a *lambda expression* λ .

Lambda Expressions

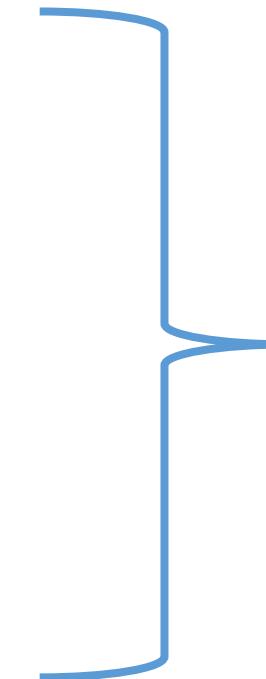
Java defines two types of lambda bodies.

- single expression lambda bodies
- block of code lambda bodies

Single Lambda Expressions(Examples)

let's look at some simple examples that put the basic lambda expression concepts into action

```
// A functional interface.  
interface MyValue  
{  
    double getValue();  
}  
// Another functional interface.  
interface MyParamValue  
{  
    double getValue(double v);  
}
```



Functional Interfaces

Single Lambda Expressions(Examples)

```
public class LambdaDemo {  
  
    public static void main(String[] args) {  
  
        MyValue myVal;  
        /*  
         * declare an interface reference Here, the lambda expression is simply a  
         * constant expression. When it is assigned to myVal, a class instance is  
         * constructed in which the lambda expression implements the getValue() method  
         * in MyValue.  
         */  
        myVal = () -> 98.6; ← A simple lambda expression  
        /*  
         * Call getValue(), which is provided by the previously assigned lambda  
         * expression.  
         */  
        System.out.println("A constant value: " + myVal.getValue());  
    }  
}
```

Single Lambda Expressions(Examples)

```
/*
 * Now, create a parameterized lambda expression and assign it to a MyParamValue
 * reference. This lambda expression returns the reciprocal of its argument.
 */
MyParamValue myPval = (n) -> 1.0 / n;
// Call getValue(v) through the myPval reference.
System.out.println("Reciprocal of 4 is " + myPval.getValue(4.0));
System.out.println("Reciprocal of 8 is " + myPval.getValue(8.0));
}
}
```

A lambda expression that has a parameter

Sample output from the program is shown here:
A constant value: 98.6
Reciprocal of 4 is 0.25
Reciprocal of 8 is 0.125

Single Lambda Expressions(Examples)

A key aspect of a functional interface is that it can be used with any lambda expression that is compatible with it.

For example, consider the following program. It defines a functional interface called **NumericTest** that declares the abstract method **test()**

This method has two **int** parameters and returns a **boolean** result. Its purpose is to determine if the two arguments passed to **test()** satisfy some condition.

Single Lambda Expressions(Examples)

```
// Use the same functional interface with three
//different lambda expressions.
// A functional interface that takes two int
//parameters and returns
// a boolean result.
```

```
interface NumericTest {
boolean test(int n, int m);
}
```

Single Lambda Expressions(Examples)

```
class LambdaDemo2 {  
    public static void main(String args[]){  
        // This lambda expression determines if one number is  
        //factor of another.  
        NumericTest isFactor = (n, d) -> (n % d) == 0;  
        if(isFactor.test(10, 2))  
            System.out.println("2 is a factor of 10");  
    }  
}
```

Single Lambda Expressions(Examples)

```
if(!isFactor.test(10, 3))  
System.out.println("3 is not a factor of 10");  
System.out.println();  
// This lambda expression returns true if the first argument  
//is less than the second.  
NumericTest lessThan = (n, m) -> (n < m);  
if(lessThan.test(2, 10)) System.out.println("2 is less than  
10");
```

Single Lambda Expressions(Examples)

```
if(!lessThan.test(10, 2)) System.out.println("10 is not  
less than 2");  
  
System.out.println();  
// This lambda expression returns true if the absolute  
// values of the arguments are equal.  
  
NumericTest absEqual = (n, m) -> (n < 0 ? -n : n)  
== (m < 0 ? -m : m);
```

Single Lambda Expressions(Examples)

```
if(absEqual.test(4, -4))  
System.out.println("Absolute values of 4 and -4 are  
equal.");  
  
if(!lessThan.test(4, -5))  
System.out.println("Absolute values of 4 and -5 are not  
equal.");  
  
System.out.println();  
}  
}
```

Single Lambda Expressions(Examples)

The output is shown here:

2 is a factor of 10

3 is not a factor of 10

2 is less than 10

10 is not less than 2

Absolute values of 4 and -4 are equal.

Absolute values of 4 and -5 are not equal.

Block Lambda Expressions

Lambda Syntax

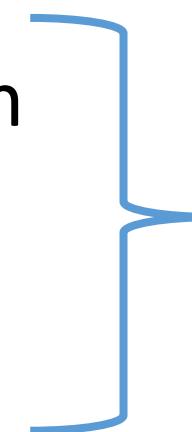
(parameters) -> expression

or

() -> expression

or

(parameters) -> {
statements; }



Single Lambda Expressions



Block Lambda Expressions

Block Lambda Expressions

- Lambdas that have block bodies are sometimes referred to as *block lambdas*.
- A block lambda expands the types of operations that can be handled within a lambda expression because it allows the body of the lambda to contain multiple statements.
- Aside from allowing multiple statements, block lambdas are used much like the expression lambdas just discussed.
- One key difference, however, is that you must explicitly use a **return** statement to return a value.

Block Lambda Expressions

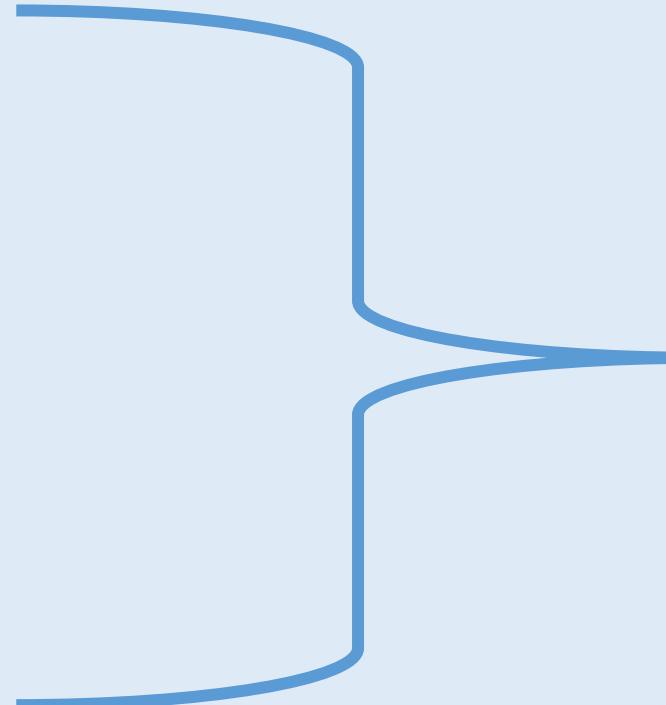
- Here is an example that uses a block lambda to find the smallest positive factor of an **int** value.
- It uses an interface called **NumericFunc** that has a method called **func()**, which takes one **int** argument and returns an **int** result. Thus, **NumericFunc** supports a numeric function on values of type **int**.

```
// A block lambda that finds the smallest positive factor
// of an int value.

interface NumericFunc {
    int func(int n);
}
```

Block Lambda Expressions

```
class BlockLambdaDemo {  
  
    public static void main(String args[]) {  
  
        // This block lambda returns the smallest positive factor of a value.  
  
        NumericFunc smallestF = (n) -> {  
            int result = 1;  
  
            // Get absolute value of n.  
  
            n = n < 0 ? -n : n;  
  
            for(int i=2; i <= n/i; i++)  
                if((n % i) == 0) {  
                    result = i;  
                    break;  
                }  
  
            return result;  
        };  
  
        System.out.println("Smallest factor of 12 is " + smallestF.func(12));  
        System.out.println("Smallest factor of 11 is " + smallestF.func(11)); } }
```



Block Lambda Expression

Generic Functional Interface

The functional interface associated with a lambda expression can be generic.

```
// Use a generic functional interface.  
  
// A generic functional interface with two parameters  
// that returns a boolean result.  
  
interface SomeTest<T>  
  
{  
  
    boolean test(T n, T m);  
  
}
```

Generic Functional Interface

```
class GenericFunctionalInterfaceDemo {  
  
    public static void main(String args[]) {  
  
        // This lambda expression determines if one integer is a factor of another.  
  
        SomeTest<Integer> isFactor = (n, d) -> (n % d) == 0;  
  
        if(isFactor.test(10, 2)) System.out.println("2 is a factor of 10");  
  
        System.out.println();  
  
        // The next lambda expression determines if one Double is a factor of another.  
  
        SomeTest<Double> isFactorD = (n, d) -> (n % d) == 0;  
  
        if(isFactorD.test(212.0, 4.0)) System.out.println("4.0 is a factor of 212.0");  
  
        System.out.println();  
    }  
}
```

Generic Functional Interface

```
// This lambda expression determines if one string is part of another.

SomeTest<String> isIn = (a, b) -> a.indexOf(b) != -1;

String str = "Generic Functional Interface";
System.out.println("Testing string: " + str);

if(isIn.test(str, "face")) System.out.println("'face' is found.");
else System.out.println("'face' not found.");

}
```

Method References

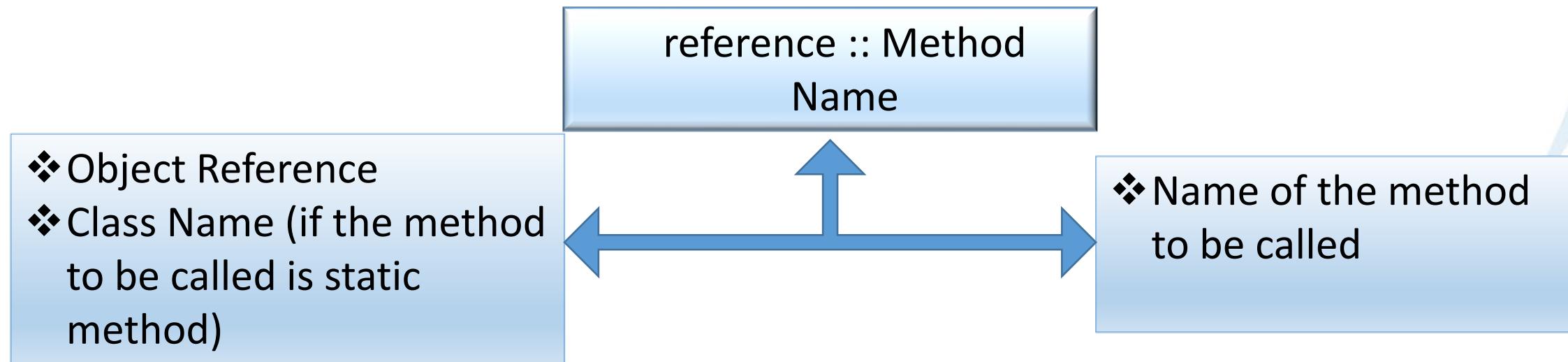
- What are method references?
- Method reference syntax
- Method Reference types
- Method reference examples

Method References

- It is a feature which is related to Lambda Expression.
- It allows us to reference constructors or methods without executing them.
- Method references and Lambda are similar in that they both require a target type that consist of a compatible functional interface.

Method References

- What are method references?
 - A new feature in Java SE 8
 - Allows to use a method as a value
- Method reference syntax



Types of Method Reference

Type	Example	Syntax
Reference to a static method	<i>ContainingClass::staticMethodName</i>	Class::staticMethodName
Reference to a constructor	<i>ClassName::new</i>	ClassName::new
Reference to an instance method of an arbitrary object of a particular type	<i>ContainingType::methodName</i>	Class::instanceMethodName
Reference to an instance method of a particular object	<i>containingObject::instanceMethodName</i>	object::instanceMethodName

1 . Reference to a Static Method

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.function.BiPredicate;
public class Numbers {
    public static boolean isMoreThanFifty(int n1, int n2) {
        return (n1 + n2) > 50;
    }
    public static List<Integer> findNumbers(List<Integer> l, BiPredicate<Integer, Integer> p)
    {
        List<Integer> newList = new ArrayList<>();
        for (Integer i : l) {
            if (p.test(i, i + 10)) {
                newList.add(i);
            }
        }
        return newList;
    }
}
```

1 . Reference to a Static Method

```
public static void main(String[] args) {  
    List<Integer> list = Arrays.asList(12, 5, 45, 18, 33, 24, 40);  
    // Using an anonymous class  
    findNumbers(list, new BiPredicate<Integer, Integer>() {  
        public boolean test(Integer i1, Integer i2) {  
            return Numbers.isMoreThanFifty(i1, i2);  
        }  
    });  
    // Using a lambda expression  
    findNumbers(list, (i1, i2) -> Numbers.isMoreThanFifty(i1, i2));  
    // Using a method reference  
    findNumbers(list, Numbers::isMoreThanFifty);  
}
```

2. Reference To Constructor

```
import java.util.*;
import java.util.function.Supplier;
public class Test {
    public static void main(String args[]) {
        // Using an anonymous class
        Supplier<List<String>> s = new Supplier<List<String>>() {
            public List<String> get() {
                return new ArrayList<String>();
            }
        };
        List<String> l = s.get();
        // Using a lambda expression
        Supplier<List<String>> s1 = () -> new ArrayList<String>();
        List<String> l1 = s1.get();
        // Using a method reference
        Supplier<List<String>> s2 = ArrayList<String>::new;
        List<String> l2 = s2.get();
        System.out.println(l + "----" + l1 + " ---- " + l2);
    }
}
```

3. Reference To an Instance Method Of An Arbitrary Object Of A Particular Type

```

import java.util.ArrayList;
import java.util.List;
import java.util.function.Function;
public class Test {
public static void main(String[] args) {
// TODO Auto-generated method stub
Test t = new Test();
List<Shipment> l = new ArrayList<Shipment>();
// Using an anonymous class
t.calculateOnShipments(l, new Function<Shipment, Double>() {
public Double apply(Shipment s) { // The object
return s.calculateWeight(); // The method
});
// Using a lambda expression
t.calculateOnShipments(l, s -> s.calculateWeight());
// Using a method reference
t.calculateOnShipments(l, Shipment::calculateWeight);
}

```

```

class Shipment {
public double calculateWeight()
{
double weight = 100;
// Calculate weight
return weight;
}
}

```

3. Reference To An Instance Method Of An Arbitrary Object Of A Particular Type

```
public List<Double> calculateOnShipments(List<Shipment> l, Function<Shipment, Double> f)
{
    List<Double> results = new ArrayList<>();
    for (Shipment s : l) {
        results.add(f.apply(s));
    }
    return results;
}
```

4. Reference To An Instance Method Of A Particular Object

```
class Car {  
    private int id;  
    private String color;  
    // More properties  
    // And getter and setters  
    public int getId() {  
        return id;  
    }  
    public void setId(int id) {  
        this.id = id;  
    }  
    public String getColor() {  
        return color;  
    }  
    public void setColor(String color) {  
        this.color = color;  
    }  
}
```

```
class Mechanic {  
    public void fix(Car c) {  
        System.out.println("Fixing car " +  
            c.getId());  
    }  
}
```

4. Reference To An Instance Method Of A Particular Object

```
import java.util.function.Consumer;

public class Test {
    public static void main(String[] args) {
        Test t = new Test();
        final Mechanic mechanic = new Mechanic();
        Car car = new Car();
        // Using an anonymous class
        t.execute(car, new Consumer<Car>() {
            public void accept(Car c) {
                mechanic.fix(c);
            }
        });
        // Using a lambda expression
        t.execute(car, c -> mechanic.fix(c));
        // Using a method reference
        t.execute(car, mechanic::fix);
    }
}
```

4. Reference To An Instance Method Of A Particular Object

```
public void execute(Car car, Consumer<Car> c)
{
    c.accept(car);
}

}
```

Another Method Reference Example

- You want to filter all the hidden files in a directory
 - You need to start writing a method that given a File will tell you whether it's hidden or not
 - there's such a method inside the File class called isHidden
 - It can be viewed as a function that takes a File and returns a boolean
 - But to use it for filtering you need to wrap it into a FileFilter object that you then pass to the File.listFiles method

Method Reference Example

```
File[] hiddenFiles= new File(".").listFiles(new  
FileFilter() {  
public boolean accept(File file) {  
return file.isHidden(); } } );
```

Filter Hidden
Files

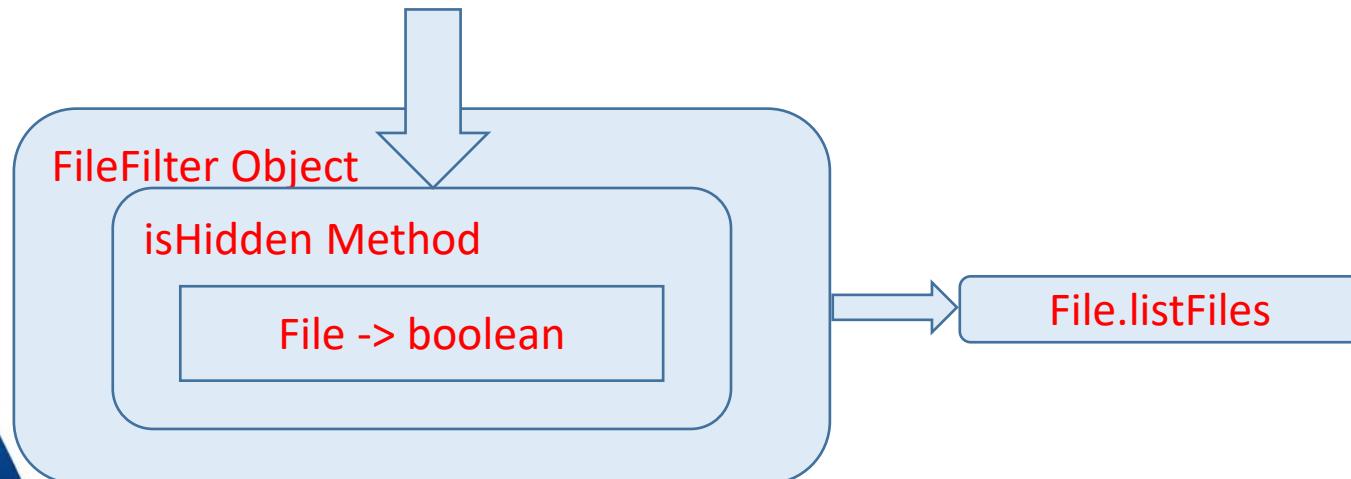
Now, in Java 8 you can rewrite that code as follows:

```
File[] hiddenFiles = new File(".").listFiles(File::isHidden);
```

Method Reference Example

Old way of filtering hidden files

```
File[] hiddenFiles= new File(".").listFiles(new FileFilter() {  
    public boolean accept(File file) {  
        return file.isHidden(); }  
});
```

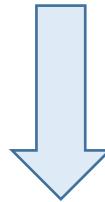


Filtering files with the `isHidden` method requires wrapping the method inside a `FileFilter` object before passing it to the `File.listFiles` method

Method Reference Example

Java 8 Style

```
File[] hiddenFiles = new File(".").listFiles(File::isHidden);
```



File.isHidden

File :: isHidden (new Syntax)



File.listFiles

In Java 8 pass the isHidden function to the listFiles method using the method reference :: Syntax

Lab Exercise-1

- Your goal is to make a method called betterString that takes two Strings and a lambda that says whether the first of the two is “better”.
- The method should return that better String; i.e., if the function given by the lambda returns true, the betterString method should return the first String, otherwise betterString should return the second String.
- *String string1 = ...;*
- *String string2 = ...;*
- *String longer = StringUtils.betterString(string1, string2, (s1, s2) -> s1.length() > s2.length());*
- *String first = StringUtils.betterString(string1, string2, (s1, s2) -> true);*

Lab Exercise-2

- Given a String, the task is to check whether a string contains only alphabets or not.
- use *isLetter()* method of *Character class*.

A.	The History and Evolution of Java	2
B.	An Overview of Java	21
C.	Data Types, Variables and Arrays	64
D.	Operators, Control Statements and String Handling	98
E.	Packages and Interfaces	163
F.	Wrapper Classes, Enumeration, Autoboxing and Annotations	216
G.	Exception Handling	256
H.	Generics and Lambda Expressions	299
I.	Java Stream API	363
J.	Java Collections	430
K.	Java Platform Module System	522

Lesson 9

Introduction to JAVA Stream API

Stream API

- Java SE 8 introduces the concept of streams.
- Streams are objects of classes that implement interface Stream (from the package *java.util.stream*) or one of the specialized stream interfaces for processing collections of int, long or double values.
- Together with lambdas, streams enable you to perform tasks on collections of elements—often from an array or collection object.
- First of all, Java 8 Streams should not be confused with Java I/O streams (ex: *FileInputStream* etc); these have very little to do with each other.

What are Streams

- streams are wrappers around a data source, allowing us to operate with that data source and making bulk processing convenient and fast.
- **A stream does not store data and, in that sense, is not a data structure. It also never modifies the underlying data source.**
- *In a nutshell, the term "MapReduce" refers to two distinct tasks. The first is the **Map** job, which takes one set of data and transforms it into another set of data, where individual elements are broken down into tuples (**key/value pairs**), while the **Reduce** job takes the output from a map as input and combines those data tuples into a smaller set of tuples.*

Stream API: Stream Pipelines

- Streams move elements through a sequence of processing steps—known as a stream pipeline—that begins with a data source (such as an array or collection), performs various intermediate operations on the data source's elements and ends with a terminal operation.
- A stream pipeline is formed by chaining method calls. Unlike collections, ***streams do not have their own storage***—once a stream is processed, it cannot be reused, because it does not maintain a copy of the original data source.

Java Stream Creation

- Let's first obtain a stream from an existing array:

```
private static Employee[] arrayOfEmps = {  
    new Employee(1, "Jeff Bezos", 100000.0),  
    new Employee(2, "Bill Gates", 200000.0),  
    new Employee(3, "Mark Zuckerberg",  
    300000.0)  
};
```

```
public static void main(String args[]){  
    Stream empStreamFromArray=Stream.of(arrayOfEmps);  
    Stream empStreamFromList =empList.stream();  
}
```

- We can also obtain a stream from an existing list:

```
private static List<Employee> empList =  
    Arrays.asList(arrayOfEmps);
```

Java Stream Creation

- we can create a stream from individual objects using *Stream.of()*:

```
Stream.of(arrayOfEmps[0],  
arrayOfEmps[1], arrayOfEmps[2]);
```

- Or simply using *Stream.builder()*:

```
Stream.Builder<Employee> empStreamBuilder  
= Stream.builder();
```

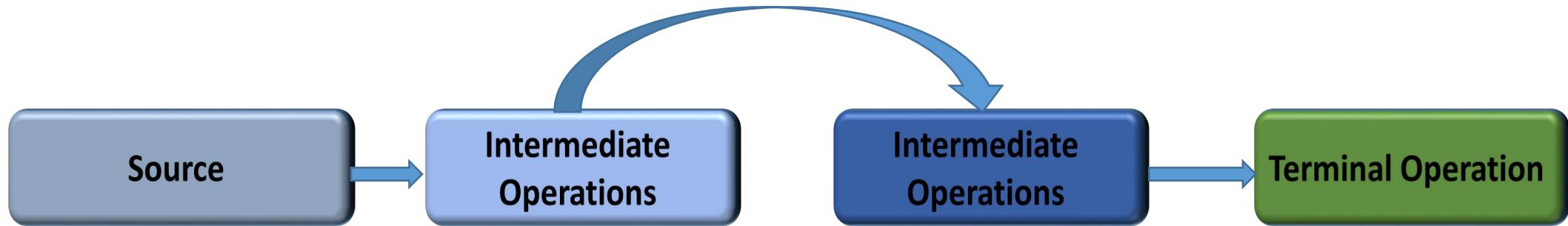
```
empStreamBuilder.accept(arrayOfEmps[0]);  
empStreamBuilder.accept(arrayOfEmps[1]);  
empStreamBuilder.accept(arrayOfEmps[2]);
```

```
Stream<Employee> empStream =  
empStreamBuilder.build();
```

Java Stream Operations

- In the next Slides we will see some common usages and operations we can perform on and with the help of the stream support in the language.

Stream API: Stream Pipelines



```
.stream()  
    .filter(b -> b.getColor() == red)  
    .mapToInt( b -> b.getWeight() )  
    .sum();
```

Stream API: Intermediate Operations

- An intermediate operation specifies tasks to perform on the stream's elements and always results in a new stream.
- Intermediate operations are lazy—they aren't performed until a terminal operation is invoked.
- This allows library developers to optimize stream-processing performance.

Stream API: Intermediate Operations

- For example, if you have a collection of 1,000,000 Person objects and you're looking for the first one with the last name "Jones", stream processing can terminate as soon as the first such Person object is found.

Stream API: Terminal Operations

- A terminal operation initiates processing of a stream pipeline's intermediate operations and produces a result.
- Terminal operations are eager—they perform the requested operation when they are called.
- The Below tables show some common intermediate and terminal operations

Stream API: Common Intermediate Stream Operations

Operation	Description
filter	Results in a stream containing only the elements that satisfy a condition.
distinct	Results in a stream containing only the unique elements.
limit	Results in a stream with the specified number of elements from the beginning of the original stream.
map	Results in a stream in which each element of the original stream is mapped to a new value (possibly of a different type). The new stream has the same number of elements as the original stream.
sorted	Results in a stream in which the elements are in sorted order. The new stream has the same number of elements as the original stream.

Stream API: Common Terminal Stream Operations

Loop

- **forEach**- Performs processing on every element in a stream (e.g., display each element).

Reduction operations

- Take all values in the stream and return a single value

Mutable reduction operations

- Create a container (such as a collection or StringBuilder)

Search operations

- Performs different search or match operations on a stream

Stream API: Common Terminal Stream Operations

Operation	Reduction operations
average	Calculates the average of the elements in a numeric stream.
count	Returns the number of elements in the stream.
max	Locates the largest value in a numeric stream.
min	Locates the smallest value in a numeric stream.
reduce	Reduces the elements of a collection to a single value using an associative accumulation function (e.g., a lambda that adds two elements).

Stream API: Common Terminal Stream Operations

Operation	Mutable Reduction operations
<code>collect</code>	Creates a new collection of elements containing the results of the stream's prior operations.
<code>toArray</code>	Creates an array containing the results of the stream's prior operations..

Stream API: Common Terminal Stream Operations

Operation	Search operations
findFirst	Finds the first stream element based on the prior intermediate operations; immediately terminates processing of the stream pipeline once such an element is found.
findAny	Finds any stream element based on the prior intermediate operations; immediately terminates processing of the stream pipeline once such an element is found.
anyMatch	Determines whether any stream elements match a specified condition; immediately terminates processing of the stream pipeline if an element matches.
allMatch	Determines whether all of the elements in the stream match a specified condition.

forEach

- *forEach()* is simplest and most common operation; it loops over the stream elements, calling the supplied function on each element.

```
empList.stream().forEach(e -> e.salaryIncrement(10.0));
```

- *This will effectively call the salaryIncrement() on each element in the empList.*
- *forEach() is a terminal operation, which means that, after the operation is performed, the stream pipeline is considered consumed, and can no longer be used. We'll talk more about terminal operations in the next slides.*

map

- *map()* produces a new stream after applying a function to each element of the original stream. The new stream could be of different type.
- The following example converts the stream of *Integers* into the stream of *Employees*:

```
public void whenMapIdToEmployees_thenGetEmployeeStream() {  
    Integer[] emplIds = { 1, 2, 3 };  
    EmployeeRepository employeeRepository =new EmployeeRepository();  
    List<Employee> employees = Stream.of(emplIds)  
        .map(employeeRepository::findById)  
        .collect(Collectors.toList());  
}
```

map

- Here, we obtain an *Integer* stream of employee ids from an array.
- Each *Integer* is passed to the function `employeeRepository::findById()` – which returns the corresponding *Employee* object; this effectively forms an *Employee* stream.

Stream API Example

- In the following example we will show how to use Stream API and lambda expressions to simplify programming tasks that you probably already performed for ***Arrays*** and ***ArrayLists***
- The Example demonstrates operations on an ***IntStream*** (package ***java.util.stream***)—a specialized stream for manipulating int values.
- The techniques shown in this example also apply to ***LongStreams*** and ***DoubleStreams*** for long and double values, respectively.

Stream API Example

```
import java.util.stream.IntStream;  
  
public class IntStreamOperations {  
  
    public static void main(String[] args) {  
  
        int[] values = {3, 10, 6, 1, 4, 8, 2, 5, 9, 7};  
  
        // display original values  
  
        System.out.print("Original values: ");  
  
        IntStream.of(values).forEach(value -> System.out.printf("%d ", value));  
  
        System.out.println();
```

Stream API Example

```
//count, min, max, sum and average of the values  
  
System.out.printf("%nCount: %d%n", IntStream.of(values).count());  
  
System.out.printf("Min: %d%n",  
IntStream.of(values).min().getAsInt());  
  
System.out.printf("Max: %d%n",  
IntStream.of(values).max().getAsInt());  
  
System.out.printf("Sum: %d%n", IntStream.of(values).sum());  
  
System.out.printf("Average: %.2f%n",  
IntStream.of(values).average().getAsDouble());
```

Returns OptionallInt which is a container object which may or may not contain an int value

Stream API Example

The identity value must be an identity for the accumulator function. This means that for all x , `accumulator.apply(identity, x)` is equal to x . The accumulator function must be an associative function.

```
//sum of squares of values with reduce method
```

```
System.out.printf("Sum of squares via reduce method:
```

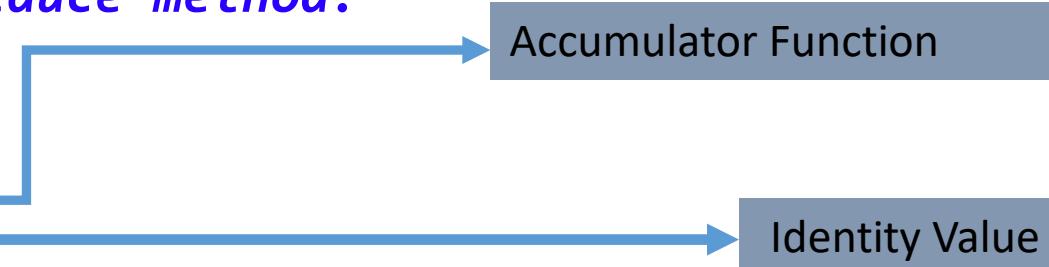
```
%d%n", IntStream.of(values)
```

```
.reduce(0, (x,y) -> x+y*y));
```

```
//product of values with reduce method
```

```
System.out.printf("Product via reduce method: %d%n", IntStream.of(values)
```

```
.reduce(1,(x,y)->x*y));
```



Stream API Example

```
//even values displayed in sorted order
System.out.printf("%nEven values displayed in sorted order: ");
IntStream.of(values).filter(value -> value % 2 == 0)
.sorted().forEach(value -> System.out.printf("%d ", value));
System.out.println();

//odd values multiplied by 10 and displayed in sorted order
System.out.printf("Odd values multiplied by 10 displayed in sorted order: ");
IntStream.of(values).filter(value -> value % 2 != 0)
.map(value -> value * 10)
.sorted()
.forEach(value -> System.out.printf("%d", value));
System.out.println();
```

Stream API Example

```
//sum range of integers from 1 to 10, exclusive
System.out.printf("%nSum of integers from 1 to 9: %d%n",
IntStream.range(1, 10).sum());

//sum range of integers from 1 to 10, inclusive
System.out.printf("Sum of integers from 1 to 10: %d%n",
IntStream.rangeClosed(1, 10).sum());

}

} // end class IntStreamOperations
```

collect

- We saw how *collect()* works in the previous example; its one of the common ways to get stuff out of the stream once we are done with all the processing.
- *collect()* performs mutable fold operations (*repackaging elements to some data structures and applying some additional logic, concatenating them, etc.*) on data elements held in the *Stream* instance.
- The strategy for this operation is provided via the *Collector* interface implementation. In the example above, we used the *toList* collector to collect all *Stream* elements into a *List* instance.

filter

- *filter()* produces a new stream that contains elements of the original stream that pass a given test (specified by a Predicate).
- In the example in the next slide, we first filter out *null* references for invalid employee ids and then again apply a filter to only keep employees with salaries over a certain threshold.

filter

```
public void whenFilterEmployees_thenGetFilteredStream() {  
    Integer[] emplIds = { 1, 2, 3, 4 };  
    List<Employee> employees = Stream.of(emplIds)  
        .map(employeeRepository::findById)  
        .filter(e -> e != null)  
        .filter(e -> e.getSalary() > 200000)  
        .collect(Collectors.toList());  
}
```

Another Stream API Example

```
List<Dish> menu = Arrays.asList(  
    new Dish("beef", false, 700, Dish.Type.MEAT),  
    new Dish("chicken", false, 400, Dish.Type.MEAT),  
    new Dish("french fries", true, 530, Dish.Type.OTHER),  
    new Dish("rice", true, 350, Dish.Type.OTHER),  
    new Dish("season fruit", true, 120, Dish.Type.OTHER),  
    new Dish("pizza", true, 550, Dish.Type.OTHER),  
    new Dish("prawns", false, 300, Dish.Type.FISH),  
    new Dish("salmon", false, 450, Dish.Type.FISH) );
```

Another Stream API Example

- Consider the following code snippets

```
List<Dish>  
vegetarianDishes = new  
ArrayList<>();  
for(Dish d: menu) {  
if(d.isVegetarian()) {  
vegetarianDishes.add(d);  
}  
}
```

Without using Stream API

```
import static  
java.util.stream.Collectors.  
toList;  
List<Dish> vegetarianDishes  
=  
menu.stream()  
.filter(Dish::isVegetarian)  
.collect(toList());
```

Using Stream API

Another Stream API Example

```
import static java.util.stream.Collectors.toList;  
  
List<String> threeHighCalorieDishNames =  
    menu.stream()  
        .filter(d ->d.getCalories() >300)  
        .map(Dish::getName)  
        .limit(3)  
        .collect(toList());  
  
System.out.println(threeHighCalorieDishNames);
```

Select
only the
first
three

Store the
result in
another
List

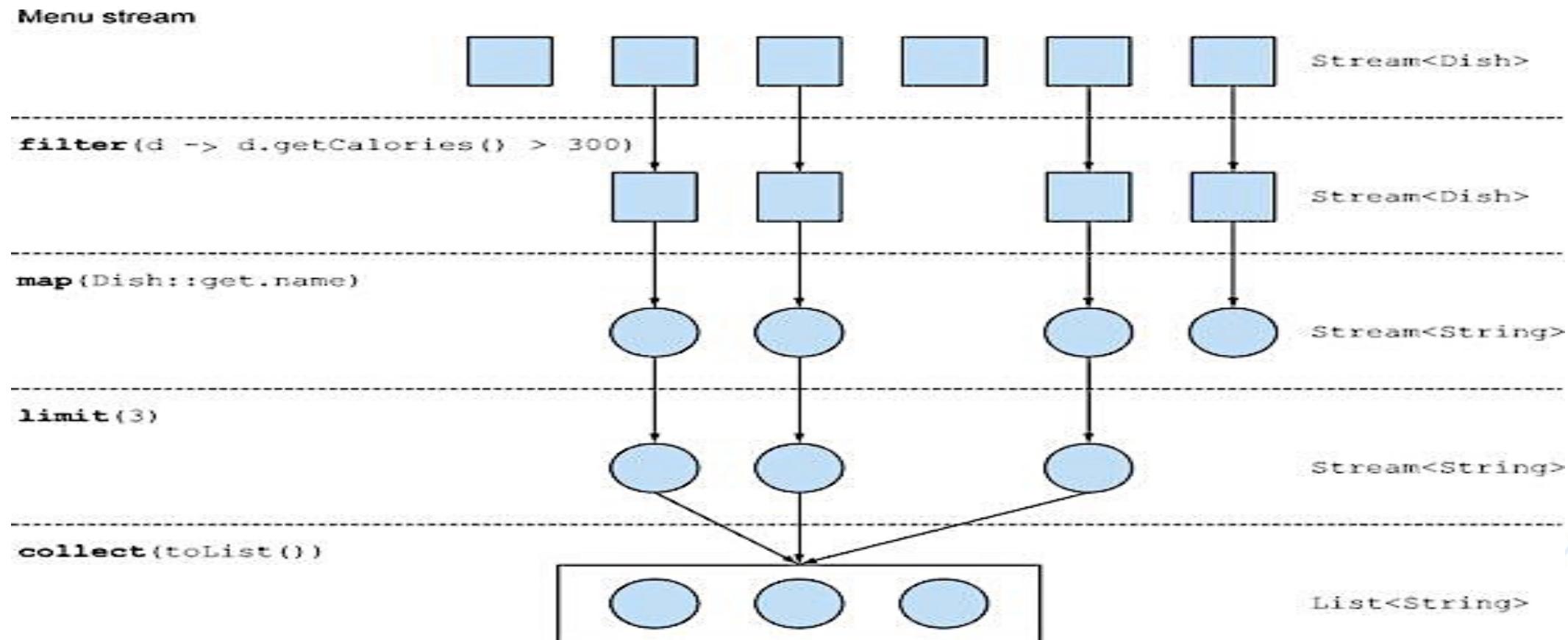
Get a Stream from menu (the list
of dishes)

Create a pipeline of operations:
first filter high-calorie dishes.

Get the names of
the dishes

The result is []

Another Stream API Example



peek

- We saw *forEach()* earlier in this section, which is a terminal operation.
- However, sometimes we need to perform multiple operations on each element of the stream before any terminal operation is applied.
- *peek()* can be useful in situations like this. Simply put, it performs the specified operation on each element of the stream and returns a new stream which can be used further.
***peek()* is an intermediate operation:**

peek

```
public void whenIncrementSalaryUsingPeek_thenApplyNewSalary()
{
    Employee[] arrayOfEmps = {
        new Employee(1, "Jeff Bezos", 100000.0),
        new Employee(2, "Bill Gates", 200000.0),
        new Employee(3, "Mark Zuckerberg", 300000.0)
    };
    List<Employee> empList = Arrays.asList(arrayOfEmps);
    empList.stream()
        .peek(e -> e.salaryIncrement(10.0))
        .peek(System.out::println)
        .collect(Collectors.toList());
}
```

Here, the first *peek()* is used to increment the salary of each employee. The second *peek()* is used to print the employees. Finally, *collect()* is used as the terminal operation.

Method Types and Pipelines

- As we've been discussing, Java stream operations are divided into intermediate and terminal operations.
- Intermediate operations such as *filter()* return a new stream on which further processing can be done.
- Terminal operations, such as *forEach()*, mark the stream as consumed, after which point it can no longer be used further.
- A stream pipeline consists of a stream source, followed by zero or more intermediate operations, and a terminal operation.

Method Types and Pipelines

- Here's a sample stream pipeline, where *empList* is the source, *filter()* is the intermediate operation and *count* is the terminal operation

```
public void whenStreamCount_thenGetElementCount() {  
    Long empCount = empList.stream()  
        .filter(e -> e.getSalary() > 200000)  
        .count();  
  
}
```

Method Types and Pipelines

- Some operations are considered **short-circuiting operations**.
- Short-circuiting operations allow computations on infinite streams to complete in finite time

```
public void whenLimitInfiniteStream_thenGetFiniteElements() {  
    Stream<Integer> infiniteStream = Stream.iterate(2, i -> i * 2);
```

```
List<Integer> collect = infiniteStream  
    .skip(3)  
    .limit(5)  
    .collect(Collectors.toList());
```

Here, we use short-circuiting operations skip() to skip first 3 elements, and limit() to limit to 5 elements from the infinite stream generated using iterate().

Lazy Evaluation

- One of the most important characteristics of Java streams is that they allow for significant optimizations through lazy evaluations.
- Computation on the source data is only performed when the terminal operation is initiated, and source elements are consumed only as needed.
- All intermediate operations are lazy, so they're not executed until a result of a processing is actually needed.

Lazy Evaluation

- For example, consider the *findFirst()* example below.
How many times is the *map()* operation performed here?

- 4 times
- Maximum 4 times

```
public void whenFindFirst_thenGetFirstEmployeeInStream() {  
    Integer[] emplIds = { 1, 2, 3, 4 };
```

```
Employee employee = Stream.of(emplIds)  
    .map(employeeRepository::findById)  
    .filter(e -> e != null)  
    .filter(e -> e.getSalary() > 100000)  
    .findFirst()  
    .orElse(null);
```

Stream performs the *map* and two *filter* operations, one element at a time.

Comparison Based Stream Operations - sorted

- Let's start with the *sorted()* operation – this sorts the stream elements based on the comparator passed we pass into it.
- For example, we can sort *Employees* based on their names:

```
public void whenSortStream_thenGetSortedStream() {  
    List<Employee> employees = empList.stream()  
        .sorted((e1, e2) -> e1.getName().compareTo(e2.getName()))  
        .collect(Collectors.toList());  
}
```

Comparison Based Stream Operations – min and max

- As the name suggests, *min()* and *max()* return the minimum and maximum element in the stream respectively, based on a comparator.
- They return an *Optional* since a result may or may not exist.

```
public void whenFindMin_thenGetMinElementFromStream() {  
    Employee firstEmp = empList.stream()  
        .min((e1, e2) -> e1.getId() - e2.getId())  
        .orElseThrow(NoSuchElementException::new);  
}
```

```
public void whenFindMax_thenGetMaxElementFromStream() {  
    Employee maxSalEmp = empList.stream()  
        .max(Comparator.comparing(Employee::getSalary))  
        .orElseThrow(NoSuchElementException::new);  
}
```

Reduction Operations

- A reduction operation (also called as fold) takes a sequence of input elements and combines them into a single summary result by repeated application of a combining operation.
- We already saw few reduction operations like *findFirst()*, *min()* and *max()*.
- Let's see the general-purpose *reduce()* operation in action.

Reduction

- The most common form of *reduce()* is:

```
T reduce(T identity, BinaryOperator<T> accumulator)
```

- where identity is the starting value and accumulator is the binary operation, we repeated apply.

```
public void  
whenApplyReduceOnStream_thenGetValue() {  
    Double sumSal = empList.stream()  
        .map(Employee::getSalary)  
        .reduce(0.0, Double::sum);  
}
```

Here, we start with the initial value of 0 and repeated apply *Double::sum()* on elements of the stream.

Advanced *collect*

- We already saw how we used *Collectors.toList()* to get the list out of the stream. Let's now see few more ways to collect elements from the stream.
 - joining
 - toSet
 - toCollection
 - summarizingDouble
 - partitioningBy
 - groupingBy
 - mapping
 - reducing

joining

- *Collectors.joining()* will insert the delimiter between the two *String* elements of the stream. It internally uses a *java.util.StringJoiner* to perform the joining operation.

```
public void whenCollectByJoining_thenGetJoinedString() {  
    String empNames = empList.stream()  
        .map(Employee::getName)  
        .collect(Collectors.joining(", "))  
        .toString();  
}
```

toSet

- We can also use `toSet()` to get a set out of stream elements:

```
public void whenCollectBySet_thenGetSet() {  
    Set<String> empNames = empList.stream()  
        .map(Employee::getName)  
        .collect(Collectors.toSet());  
}
```

toCollection

- We can use *Collectors.toCollection()* to extract the elements into any other collection by passing in a *Supplier<Collection>*. We can also use a constructor reference for the *Supplier*:

```
public void  
whenToVectorCollection_thenGetVector() {  
    Vector<String> empNames = empList.stream()  
        .map(Employee::getName)  
        .collect(Collectors.toCollection(Vector::new));  
}
```

Here, an empty collection is created internally, and its *add()* method is called on each element of the stream.

summarizingDouble

- *summarizingDouble()* is another interesting collector – which applies a double-producing mapping function to each input element and returns a special class containing statistical information for the resulting values:

```
public void whenApplySummarizing_thenGetBasicStats() {  
    DoubleSummaryStatistics stats = empList.stream()  
        .collect(Collectors.summarizingDouble(Employee::getSalary));  
  
    DoubleSummaryStatistics stats1 = empList.stream()  
        .mapToDouble(Employee::getSalary)  
        .summaryStatistics();  
}
```

summarizingDouble

```
ArrayList<Person> people = new ArrayList<>();
people.add(new Person(78.5));
people.add(new Person(87.5));
people.add(new Person(65.5));
people.add(new Person(94.5));
DoubleSummaryStatistics stats = people.stream()
    .collect(Collectors.summarizingDouble(Person::getWeight));
```

This computes, in a single pass, the count of people, as well as the minimum, maximum, sum, and average of their weights.

partitioningBy

- We can partition a stream into two – based on whether the elements satisfy certain criteria or not.
- Let's split our List of numerical data, into even and odds:

```
public void whenStreamPartition_thenGetMap() {  
    List<Integer> intList = Arrays.asList(2, 4, 5, 6, 8);  
    Map<Boolean, List<Integer>> isEven = intList.stream().collect(  
        Collectors.partitioningBy(i -> i % 2 == 0));  
}
```

- Here, the stream is partitioned into a Map, with even and odds stored as true and false keys.

groupingBy

- *groupingBy()* offers advanced partitioning – where we can partition the stream into more than just two groups.
- It takes a classification function as its parameter.
- This classification function is applied to each element of the stream.
- The value returned by the function is used as a key to the map that we get from the *groupingBy* collector.

```
public void whenStreamGroupingBy_thenGetMap() {  
    Map<Character, List<Employee>> groupByAlphabet = empList.stream().collect(  
        Collectors.groupingBy(e -> new Character(e.getName().charAt(0))));  
}
```

- we grouped the employees based on the initial character of their first name.

mapping

- *groupingBy()* discussed in the previous slide, groups elements of the stream with the use of a *Map*.
- However, sometimes we might need to group data into a type other than the element type.
- Here's how we can do that; we can use *mapping()* which can actually adapt the collector to a different type – using a mapping function:

```
public void whenStreamMapping_thenGetMap() {  
    Map<Character, List<Integer>> idGroupedByAlphabet = empList.stream().collect(  
        Collectors.groupingBy(e -> new Character(e.getName().charAt(0)),  
        Collectors.mapping(Employee::getId, Collectors.toList())));  
}
```

- Here *mapping()* maps the stream element *Employee* into just the employee id – which is an *Integer* – using the *getId()* mapping function.
- These ids are still grouped based on the initial character of employee first name.

reducing

- *reducing()* is most useful when used in a multi-level reduction, downstream of *groupingBy()* or *partitioningBy()*.
- To perform a simple reduction on a stream, use *reduce()* instead.

```
public void whenStreamGroupingAndReducing_thenGetMap() {  
    Comparator<Employee> byNameLength =  
        Comparator.comparing(Employee::getName);  
  
    Map<Character, Optional<Employee>> longestNameByAlphabet =  
        empList.stream().collect(  
            Collectors.groupingBy(e -> new Character(e.getName().charAt(0)),  
                Collectors.reducing(BinaryOperator.maxBy(byNameLength))));  
}
```

File Write Operations using streams

- Here we use *forEach()* to write each element of the stream into the file by calling *PrintWriter.println()*.

```
public void whenStreamToFile_thenGetFile() throws IOException {
    String[] words = {
        "hello",
        "refer",
        "world",
        "level"
    };
    try (PrintWriter pw = new PrintWriter(
        Files.newBufferedWriter(Paths.get(fileName)))) {
        Stream.of(words).forEach(pw::println);
    }
}
```

File Read Operations using streams

- Here *Files.lines()* returns the lines from the file as a *Stream* which is consumed by the *getPalindrome()* for further processing.
- *getPalindrome()* works on the stream, completely unaware of how the stream was generated. This also increases code reusability and simplifies unit testing.

```
public void whenFileToStream_thenGetStream(String fileName) throws IOException {  
    List<String> str = getPalindrome(Files.lines(Paths.get(fileName)), 5);  
  
}  
private List<String> getPalindrome(Stream<String> stream, int length) {  
    return stream.filter(s -> s.length() == length)  
        .filter(s -> s.compareToIgnoreCase(  
            new StringBuilder(s).reverse().toString()) == 0)  
        .collect(Collectors.toList());  
}
```

Parallel data processing and performance

- One of the most important benefit of Stream API is the possibility of executing a pipeline of operations on collections that automatically makes use of the multiple cores on your computer.

Parallel data processing and performance

- Processing a collection of data in parallel was extremely cumbersome.
 - ❖ First, you needed to explicitly split the data structure containing your data into subparts.
 - ❖ Second, you needed to assign each of these subparts to a different thread.
 - ❖ Third, you needed to synchronize them opportunely to avoid unwanted race conditions, wait for the completion of all threads, and finally combine the partial results.
 - ❖ Java 7 introduced a framework called fork/join to perform these operations more consistently and in a less error-prone way.

Parallel data processing and performance

- In the following, you'll discover how the Stream interface gives you the opportunity to execute operations in parallel on a collection of data without much effort. It lets you declaratively turn a sequential stream into a parallel one.
- Moreover, you'll see how Java can make this magic happen or, more practically, how parallel streams work under the hood by employing the fork/join framework introduced in Java 7.

Parallel data processing and performance

- it's important to know how parallel streams work internally, because if you ignore this aspect, you could obtain unexpected (and very likely wrong) results by misusing them.
- The way a parallel stream gets divided into chunks, before processing the different chunks in parallel, can in some cases be the origin of these incorrect and apparently unexplainable results.
- For this reason, you can take control of this splitting process by implementing and using your own *Spliterator*.

Parallel Streams

- we briefly mentioned that the Stream interface allows you to process its elements in parallel in a very convenient way: it's possible to turn a collection into a parallel stream by invoking the method parallel on the collection source.
 - ❖ *A parallel stream is a stream that splits its elements into multiple chunks, processing each chunk with a different thread.*

you can automatically partition the workload of a given operation on all the cores of your multicore processor and keep all of them equally busy. Let's experiment with this idea by using a simple example.

Parallel Streams

- Let's suppose you need to write a method accepting a number n as argument and returning the sum of all the numbers from 1 to the given argument.
- A straightforward (perhaps naive) approach is to generate an infinite stream of numbers, limiting it to the passed number, and then reduce the resulting stream with a ***BinaryOperator*** that just sums two numbers, as follows:

```
public static long sequentialSum(long n) {  
    return Stream.iterate(1L, i-> i+1)  
        .limit(n)  
        .reduce(0L, Long::sum);  
}
```

Parallel Streams

- This operation seems to be a good candidate to leverage parallelization, especially for large values of n. But where do you start?
- Do you synchronize on the result variable?
- How many threads do you use?
- Who does the generation of numbers?
- Who adds them up?

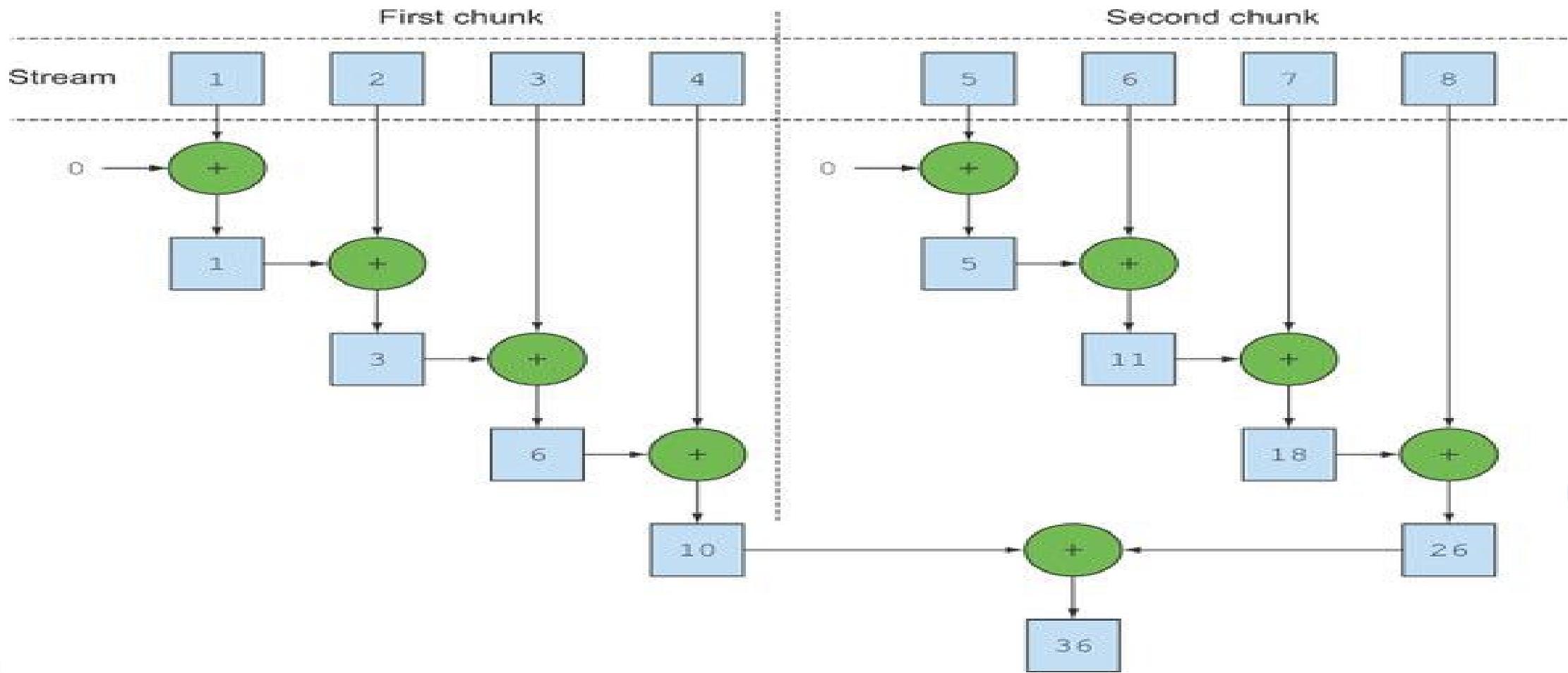
Parallel Streams

- Don't worry about all of this. It's a much simpler problem to solve if you adopt parallel streams!
- You can make the former functional reduction process (that is, summing) run in parallel by turning the stream into a parallel one; call the method parallel on the sequential stream:

Parallel Streams

```
public static long sequentialSum(long n) {  
    return Stream.iterate(1L, i-> i+1)  
        .limit(n)  
        .parallel()  
        .reduce(0L, Long::sum);  
}
```

Parallel Streams



Parallel Streams Example

```
import java.time.Duration;
import java.time.LocalTime;
import java.util.stream.IntStream;
import java.util.stream.Stream;
public class ParallelExample {
    public static void main(String[] args) {
        LocalTime startTime = LocalTime.now ();
        long count = Stream.iterate (0, n -> n + 1)
            .limit (1_000_000)
            // .parallel() //with this 23s, without this 1m 14s on intel core i7 8th generation with 32 GB memory
            .filter (ParallelExample::isPrime)
            .peek (x -> System.out.format ("%s\t", x))
            .count ();
        LocalTime endTime = LocalTime.now ();
        Duration duration = Duration.between (startTime, endTime);
        System.out.println ("\nTotal: " + count);
        System.out.println ("Computed in " + duration.getSeconds () + " seconds");
    }
    public static boolean isPrime(int number) {
        if (number <= 1) return false;
        return !IntStream.rangeClosed (2, number / 2).anyMatch (i -> number % i == 0);
    }
}
```

Lab Exercise

- **World Countries:** There are two domain classes: **Country** and **City**. Each **city** belongs to a **country** defined by the *attribute*, **countryCode**. Each **country** has a unique **code** and has many **cities**.
- The Country class has the following attributes
- The City class has the following attributes

```
private int id;
private String name;
private int population;
private String countryCode;
```

- Create the above two classes and then create a java application to find the following:
 - Find the highest populated city of each country
 - Find the most populated city of each continent
 - Find the highest populated capital city

A.	The History and Evolution of Java	2
B.	An Overview of Java	21
C.	Data Types, Variables and Arrays	64
D.	Operators, Control Statements and String Handling	98
E.	Packages and Interfaces	163
F.	Wrapper Classes, Enumeration, Autoboxing and Annotations	216
G.	Exception Handling	256
H.	Generics and Lambda Expressions	299
I.	Java Stream API	363
J.	Java Collections	430
K.	Java Platform Module System	522

Lesson 10

Java Collection Framework

Java Collections

Introduction

Introduction to Java Collection Framework

- The data structures that you choose can make a big difference when you try to implement methods in a natural style or are concerned with performance.
 - Do you need to search quickly through thousands (or even millions) of sorted items?
 - Do you need to rapidly insert and remove elements in the middle of an ordered sequence?
 - Do you need to establish associations between keys and values?

Introduction to Java Collection Framework

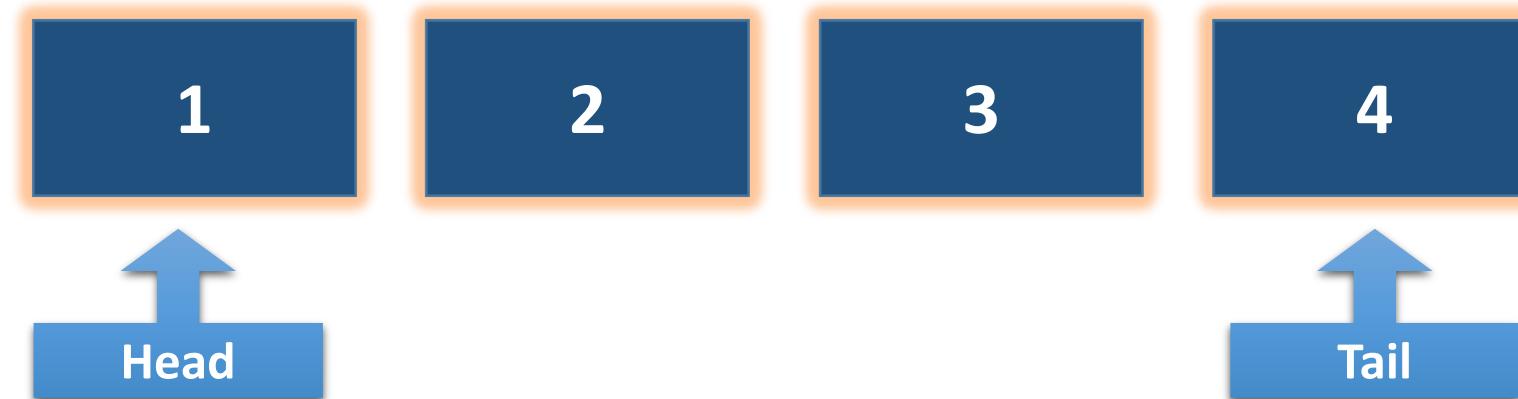
- In this lesson we will show how the Java library can help you accomplish the traditional data structuring needed for serious programming.
- We will show how to use the collection classes in the Java standard library

The Java Collections Framework

- The initial release of Java supplied only a small set of classes for the most useful data structures: ***Vector*, *Stack*, *Hashtable***
- With the introduction of Java SE 1.2, the designers felt that the time had come to rollout a full-fledged set of data structures.
- They wanted the library to be small and easy to learn.
- We will explore the basic design of the Java collections framework, show you how to put it to work, and explain the reasoning behind some of the more controversial features.

Separating Collection Interfaces and Implementation

- The Java collection library separates *interfaces* from *implementations*.
 - Let us look at that separation with a familiar data structure, the *queue*.
 - You use a queue when you need to collect objects and retrieve them in a “first in, first out” fashion



Separating Collection Interfaces and Implementation

- A *queue interface* specifies that you can add elements at the tail or the end of the queue, remove them at the head, and find out how many elements are in the queue.

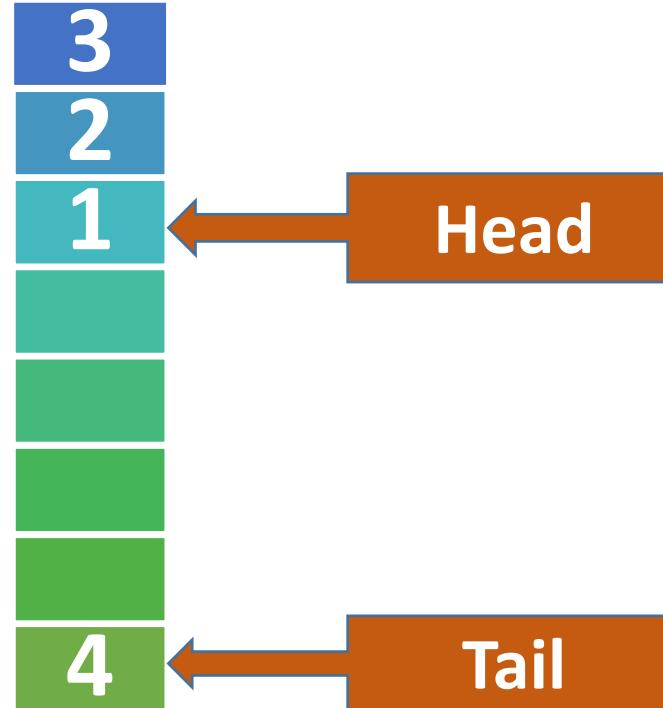
```
// a simplified form of the interface in the standard library
interface Queue<E> {

    void add(E element);
    E remove();
    int size();
}
```

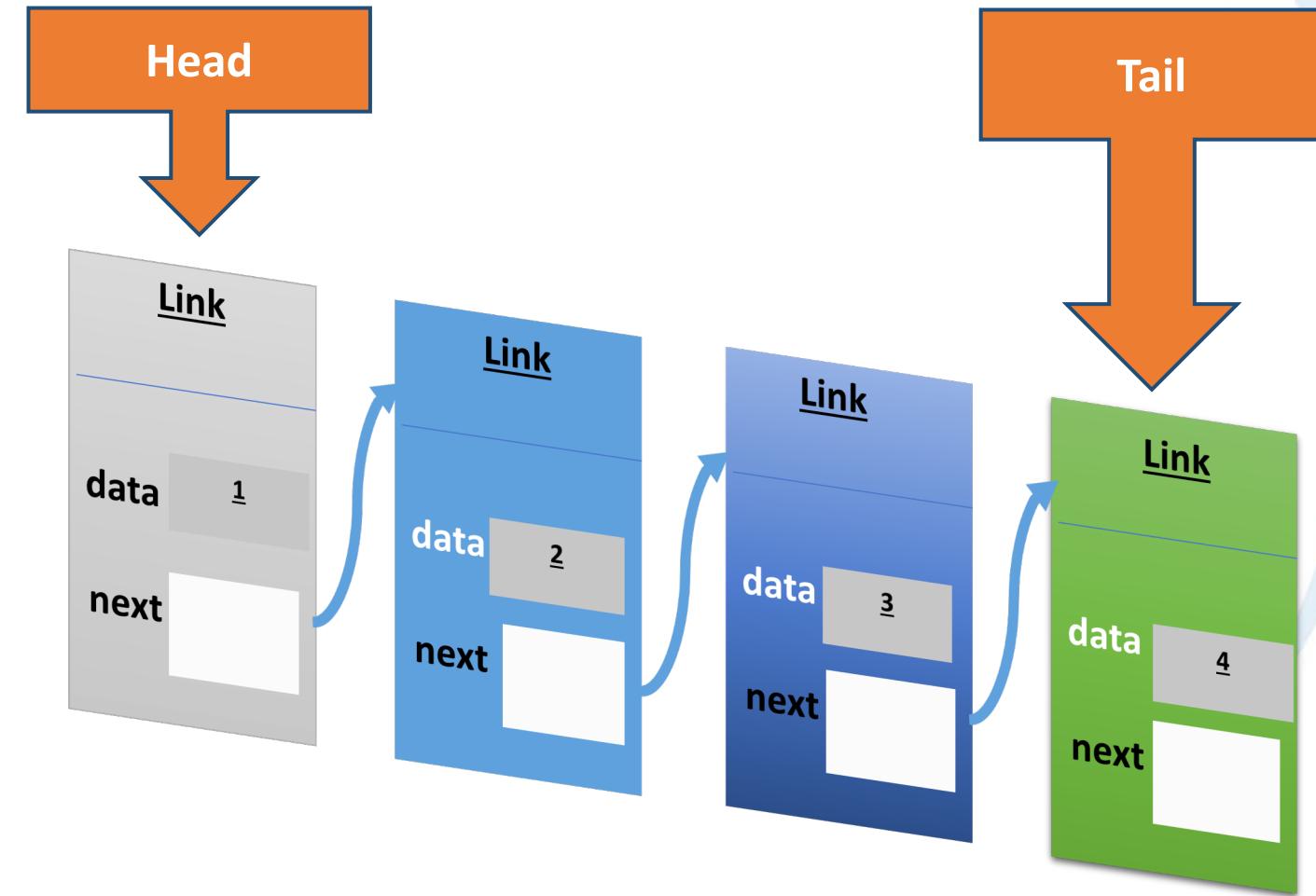
Separating Collection Interfaces and Implementation

- The interface tells you nothing about how the queue is implemented.
- Of the two common implementations of a queue, one uses a ***circular array*** and one uses a ***linked list***
- Each implementation can be expressed by a class that implements the Queue interface.

Separating Collection Interfaces and Implementation



Circular Array

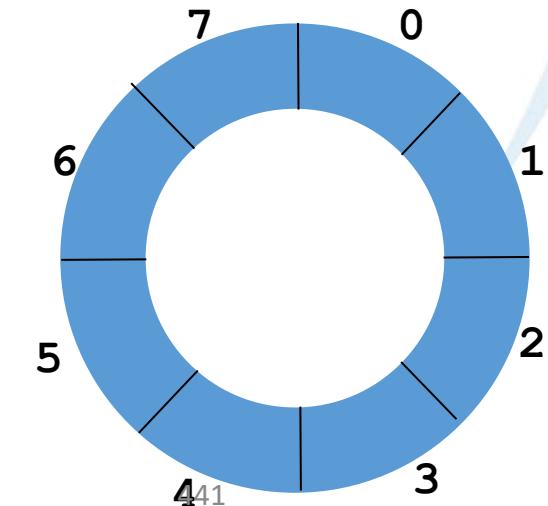


Linked List

Separating Collection Interfaces and Implementation

```
// not an actual library class
class CircularArrayQueue<E> implements Queue<E> {
    private int head;
    private int tail;
    private E[] elements;
    CircularArrayQueue(int capacity) {
    }
    public void add(E element) {
    }
    public E remove() {
        return null;
    }
    public int size() {
        return head;
    }
}
```

The idea of a circular array is that the end of the array “wraps around” to the start of the array



Separating Collection Interfaces and Implementation

```
class LinkedListQueue<E> implements Queue<E> {  
    private Link head;  
    private Link tail;  
    LinkedListQueue() {  
    }  
    public void add(E element) {  
    }  
    public E remove() {  
    }  
    public int size() {  
    } }
```

Separating Collection Interfaces and Implementation

- When you use a queue in your program, you don't need to know which implementation is actually used once the collection has been constructed.
- Therefore, it makes sense to use the concrete class *only* when you construct the collection object.
- Use the *interface type* to hold the collection reference.
`Queue<Customer> expressLane = new CircularArrayList<>(100);
expressLane.add(new Customer("Harry"));`
- With this approach, if you change your mind, you can easily use a different implementation.

Separating Collection Interfaces and Implementation

- Why would you choose one implementation over another?
- A circular array is somewhat more efficient than a linked list, so it is generally preferable.
- The circular array is a *bounded* collection—it has a finite capacity.
- If you don't have an upper limit on the number of objects that your program will collect, you may be better off with a linked list implementation after all.

The Collection Interface

- The fundamental interface for collection classes in the Java library is the ***Collection*** interface.
- The interface has two fundamental methods:
- There are several methods in addition to these two; we will discuss them later.

```
public interface Collection<E>
{
    boolean add(E element);
    Iterator<E> iterator();
    ...
}
```

The Collection Interface

- The add method adds an element to the collection.
- The add method returns true if adding the element actually changes the collection, and false if the collection is unchanged.
- The ***iterator*** method returns an object that implements the ***Iterator*** interface. You can use the iterator object to visit the elements in the collection one by one.

The Iterator Interface

- The Iterator interface has four methods:

```
public interface Iterator<E>
{
    E next();
    boolean hasNext();
    void remove();
    default void forEachRemaining(Consumer<? super E>
        action){...}
}
```

The Iterator Interface

- By repeatedly calling the next method, you can visit the elements from the collection one by one.
- However, if you reach the end of the collection, the next method throws a **NoSuchElementException**.
- Therefore, you need to call the hasNext method before calling next.

```
Collection<String> c = . . .;
Iterator<String> iter = c.iterator();
while (iter.hasNext())
{
    String element = iter.next();
    do something with element
}
```

The Iterator Interface

- As of Java SE 8, you don't even have to write a loop. You can call the ***forEachRemaining*** method with a lambda expression that consumes an element.
- The lambda expression is invoked with each element of the iterator, until there are none left.

iterator.forEachRemaining(element -> do something with element);

- The order in which the elements are visited depends on the collection type.

The Iterator Interface

- There is an important conceptual difference between iterators in the Java collections library and iterators in other libraries.
- In traditional collections libraries iterators are modeled after array indexes.
- Given such an iterator, you can look up the element that is stored at that position.
- However, the Java iterators do not work like that.
- The lookup and position change are tightly coupled. The only way to look up an element is to call next, and that lookup advances the position.

The Iterator Interface

- Think of Java iterators as being *between elements*.
- When you call `next`, the iterator *jumps over* the next element, and it returns a reference to the element that it just passed.
- The `remove` method of the Iterator interface removes the element that was returned by the last call to `next`.
- There is a dependency between the calls to the `next` and `remove` methods.
- It is illegal to call `remove` if it wasn't preceded by a call to `next`. If you try, an `IllegalStateException` is thrown.

The Iterator Interface

Initial ListIterator Position



After Calling next()



After Inserting J



A Conceptual View of the List Iterator

Generic Utility Methods

- The Collection and Iterator interfaces are generic, which means you can write utility methods that operate on any kind of collection.
- The designers of the Java library decided that some utility methods are so useful that the library should make them available.
- In fact, the Collection interface declares quite a few useful methods that all implementing classes must supply.

Generic Utility Methods

```
int size()  
boolean isEmpty()  
boolean contains(Object obj)  
boolean containsAll(Collection<?> c)  
boolean equals(Object other)  
boolean addAll(Collection<? extends E> from)  
boolean remove(Object obj)  
boolean removeAll(Collection<?> c)  
void clear()  
boolean retainAll(Collection<?> c)  
Object[] toArray()  
<T> T[] toArray(T[] arrayToFill)  
default boolean removeIf(Predicate<? super E> filter)
```

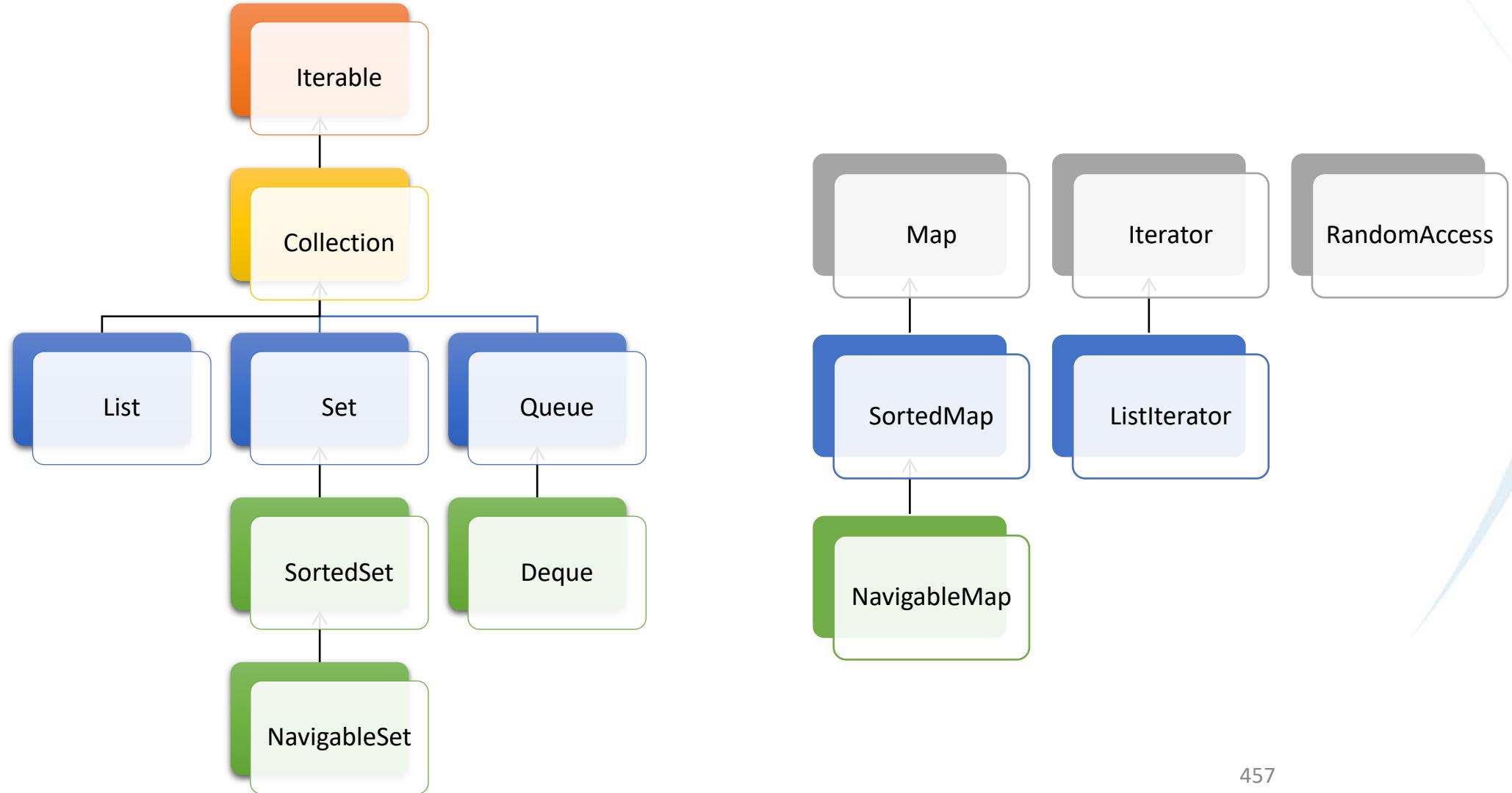
Generic Utility Methods

- To make life easier for implementers, the library supplies a class ***AbstractCollection*** that leaves the fundamental methods ***size*** and ***iterator*** abstract but implements the routine methods in terms of them.
- A concrete collection class can now extend the ***AbstractCollection*** class.
- With Java SE 8, this approach is a bit outdated. It would be nicer if the methods were default methods of the Collection interface.
- This has not happened. However, several default methods have been added. Most of them deal with streams.

Interfaces in the Collections Framework

- The Java collections framework defines a number of interfaces for different types of collections.
- There are two fundamental interfaces for collections:
Collection and ***Map***.
- you insert elements into a collection with a method
boolean add(E element)
- However, maps hold key/value pairs, and you use the put method to insert them:
V put(K key, V value)

Interfaces in the Collections Framework



Interfaces in the Collections Framework

- A List is an *ordered collection*. Elements are added into a particular position in the container.
- The ***ListIterator*** interface is a subinterface of Iterator. It defines a method for adding an element before the iterator position.
- The Set interface is identical to the Collection interface, but the behavior of the methods is more tightly defined. ***The add method of a set should reject duplicates.***

Interfaces in the Collections Framework

- The ***SortedSet*** and ***SortedMap*** interfaces expose the comparator object used for sorting, and they define methods to obtain views of subsets of the collections.
- Java SE 6 introduced interfaces ***NavigableSet*** and ***NavigableMap*** that contain additional methods for searching and traversal in sorted sets and maps.
- The ***TreeSet*** and ***TreeMap*** classes implement these interfaces.

Concrete Collections

- The following slides show the collections in the Java library and briefly describes the purpose of each collection class.
- All classes in the below table implement the ***Collection*** interface, with the exception of the classes with names ending in ***Map***. Those classes implement the Map interface instead.

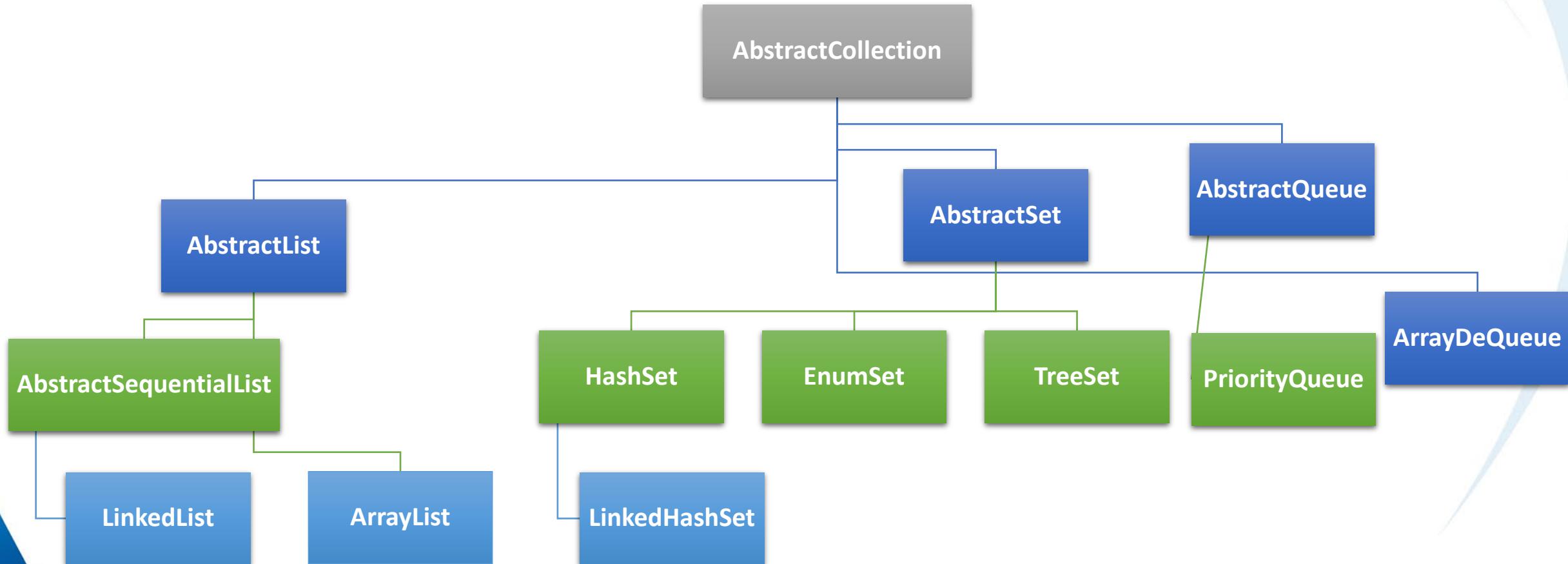
Classes in the collections framework

Collection Type	Description
ArrayList	An indexed sequence that grows and shrinks dynamically
LinkedList	An ordered sequence that allows efficient insertion and removal at any location
ArrayDeque	A double-ended queue that is implemented as a circular array
HashSet	An unordered collection that rejects duplicates
TreeSet	A sorted set
EnumSet	A set of enumerated type values
LinkedHashSet	A set that remembers the order in which elements were inserted
PriorityQueue	A collection that allows efficient removal of the smallest element

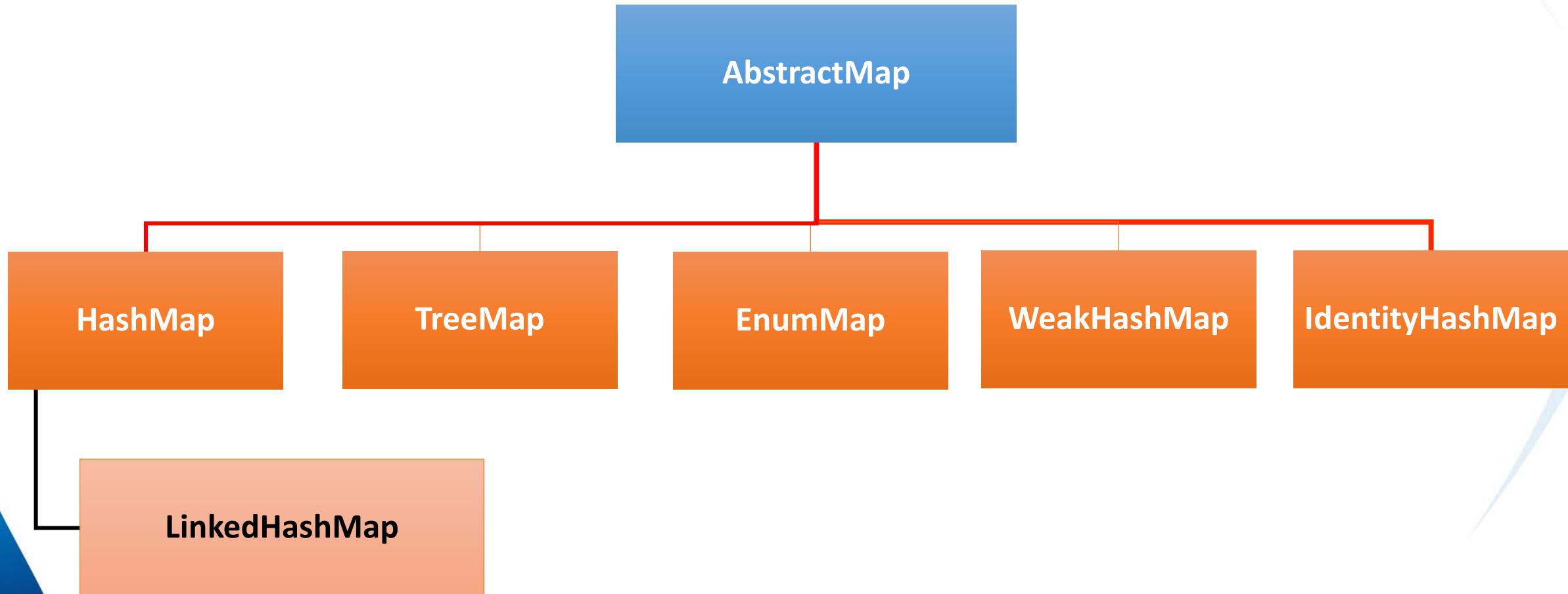
Classes in the collections framework

Collection Type	Description
HashMap	A data structure that stores key/value associations
TreeMap	A map in which the keys are sorted
EnumMap	A map in which the keys belong to an enumerated type
LinkedHashMap	A map that remembers the order in which entries were added
WeakHashMap	A map with values that can be reclaimed by the garbage collector if they are not used elsewhere
IdentityHashMap	A map with keys that are compared by ==, not equals

Classes in the Collections framework



Classes in the collections framework



Linked Lists

- Arrays and ArrayList are used frequently when we are developing different applications.
- However, arrays and array lists suffer from a major drawback.
- Removing an element from the middle of an array is expensive since all array elements beyond the removed one must be moved toward the beginning of the array. The same is true for inserting elements in the middle.
- Another well-known data structure, the *linked list*, solves this problem.

Linked Lists

- While an array stores object references in consecutive memory locations, a linked list stores each object in a separate *link*.
- Each link also stores a reference to the next link in the sequence.
- In the Java programming language, all linked lists are actually ***doubly linked***; that is, each link also stores a reference to its predecessor
- Removing an element from the middle of a linked list is an inexpensive operation—only the links around the element to be removed need to be updated

Linked Lists

- The following code example adds three elements and then removes the second one:

```
// LinkedList implements List
List<String> staff = new LinkedList<>();
staff.add("First");
staff.add("Second");
staff.add("Third");
Iterator iter = staff.iterator();
String first = iter.next(); // visit first element
String second = iter.next(); // visit second element
iter.remove(); // remove last visited element
```

Linked Lists

- A linked list is an *ordered collection* in which the position of the objects matters.
- The ***LinkedList.add*** method adds the object to the end of the list. But you will often want to add objects somewhere in the middle of a list.
- This position dependent add method is the responsibility of an iterator, since iterators describe positions in collections.
- Using iterators to add elements makes sense only for collections that have a natural ordering.

Linked Lists

- Therefore, there is no add method in the Iterator interface.
Instead, the collections library supplies a subinterface *ListIterator* that contains an add method:

```
interface ListIterator<E> extends Iterator<E>
{
    void add(E element);
    E previous()
    boolean hasPrevious()
}
```

Linked Lists

- The `listIterator` method of the `LinkedList` class returns an iterator object that implements the `ListIterator` interface.

ListIterator<String> iter = staff.listIterator();

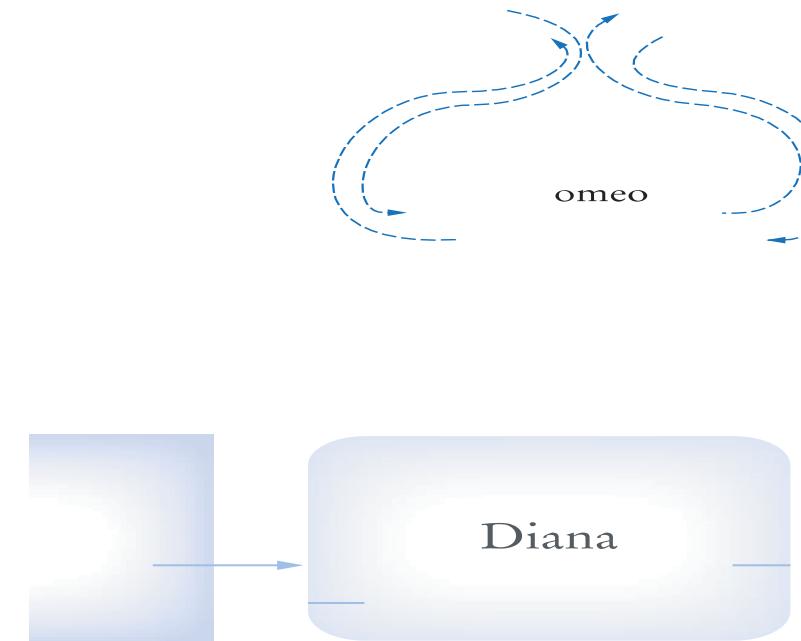
- The `add` method adds the new element *before* the iterator position.
- For example, the following code skips past the first element in the linked list and adds “Before Second” before the second element

Linked Lists

```
List<String> numbers = new LinkedList<>();  
numbers.add("First");  
numbers.add("Second");  
numbers.add("Third");  
ListIterator<String> iter = numbers.listIterator();  
iter.next(); // skip past first element  
iter.add("Before Second");
```

Linked Lists Operations

- **Efficient** Operations
 - **Insertion** of a node
 - Find the elements it goes between
 - Remap the references
 - **Removal** of a node
 - Find the element to remove
 - Remap neighbor's references
 - Visiting all elements **in order**
- **Inefficient** Operations
 - Random access



Linked Lists Concurrency

- As you might imagine, if an iterator traverses a collection while another iterator is modifying it, confusing situations can occur.
- For example, suppose an iterator points before an element that another iterator has just removed.
- The iterator is now invalid and should no longer be used. The linked list iterators have been designed to detect such modifications.
- If an iterator finds that its collection has been modified by another iterator or by a method of the collection itself, it throws a ***ConcurrentModificationException***.

Linked Lists Concurrency

```
List<String> list = . . .;  
ListIterator<String> iter1 = list.listIterator();  
ListIterator<String> iter2 = list.listIterator();  
iter1.next();  
iter1.remove();  
iter2.next(); // throws ConcurrentModificationException
```

- To avoid concurrent modification exceptions, follow this simple rule:
 - You can attach as many iterators to a collection as you like, provided that all of them are only readers.
 - Alternatively, you can attach a single iterator that can both read and write.

Linked Lists Example

- The following program puts linked lists to work.
- It simply creates two lists, merges them, then removes every second element from the second list, and finally tests the removeAll method.

Linked Lists Example

```
import java.util.*;  
  
public class LinkedListTest  
{  
    public static void main(String[] args)  
    {  
        List<String> a = new LinkedList<>();  
        a.add("A");  
        a.add("B");  
        a.add("C");  
        List<String> b = new LinkedList<>();  
        b.add("D");  
        b.add("E");  
        b.add("F");  
        b.add("G");  
    }  
}
```

Linked Lists Example

```
// merge the words from b into a
ListIterator<String> aIter = a.listIterator();
Iterator<String> bIter = b.iterator();
while (bIter.hasNext()){
    if (aIter.hasNext()) aIter.next();
    aIter.add(bIter.next());
}
System.out.println(a);
// remove every second word from b
bIter = b.iterator();
while (bIter.hasNext()){
    bIter.next(); // skip one element
    if (bIter.hasNext())
    {
        bIter.next(); // skip next element
        bIter.remove(); // remove that element
    }
}
```

Linked Lists Example

```
System.out.println(b);
// bulk operation: remove all words in b from a
a.removeAll(b);
System.out.println(a);
}
```

```
[A, D, B, E, C, F, G]
```

```
[D, F]
```

```
[A, B, E, C, G]
```

Maps

- A set is a collection that lets you quickly find an existing element.
- However, to look up an element, you need to have an exact copy of the element to find.
- That isn't a very common lookup—usually, you have some key information, and you want to look up the associated element.
- The *map* data structure serves that purpose.
- A map stores key/value pairs. You can find a value if you provide the key.

Basic Map Operations

- The Java library supplies two general-purpose implementations for maps: ***HashMap*** and ***TreeMap***. Both classes implement the Map interface.
- A hash map hashes the keys, and a tree map uses an ordering on the keys to organize them in a search tree.
- The hash or comparison function is applied *only to the keys*. The values associated with the keys are not hashed or compared.

Basic Map Operations

- Here is how you set up a hash map for storing employees:

```
//HashMap implements Map
Map<String, Employee> staff = new HashMap<>();
Employee harry = new Employee("Harry Hacker");
staff.put("987-98-9996", harry);
```

- Whenever you add an object to a map, you must supply a key as well. In our case, the key is a string, and the corresponding value is an Employee object.

Basic Map Operations

- To retrieve an object, you must use (and, therefore, remember) the key.

```
String id = "987-98-9996";  
e = staff.get(id); // gets harry
```

- If no information is stored in the map with the particular key specified, *get* returns *null*.

Basic Map Operations

- The null return value can be inconvenient. Sometimes, you have a good default that can be used for keys that are not present in the map. Then use the ***getOrDefault*** method.

Map<String, Integer> scores = . . . ;

int score = scores.getOrDefault(id, 0); // Gets 0 if the id is not present

- Keys must be unique. You cannot store two values with the same key.
- If you call the ***put*** method twice with the same key, the second value replaces the first one.
- In fact, ***put*** returns the previous value associated with its key parameter.

Basic Map Operations

- The remove method removes an element with a given key from the map.
- The size method returns the number of entries in the map.
- The easiest way of iterating over the keys and values of a map is the ***forEach*** method.

- Provide a lambda expression that receives a key and a value. That expression is invoked for each map entry in turn.

```
scores.forEach((k, v) ->  
    System.out.println("key=" + k + ", value=" + v));
```

Map Example

- We first add key/value pairs to a map.
- Then, we remove one key from the map, which removes its associated value as well.
- Next, we change the value that is associated with a key and call the get method to look up a value. Finally, we iterate through the entry set.

Map Example

```
public class Employee {  
    private String name;  
    public Employee(String name) {  
        // TODO Auto-generated constructor stub  
        this.setName(name);  
    }  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

Map Example

```
import java.util.*;  
public class MapTest {  
public static void main(String[] args) {  
Map<String, Employee> staff = new HashMap<>();  
staff.put("144-25-5464", new Employee("Amr"));  
staff.put("567-24-2546", new Employee("Mohsen"));  
staff.put("157-62-7935", new Employee("Medhat"));  
staff.put("456-62-5527", new Employee("Iman"));
```

Map Example

```
// print all entries
System.out.println(staff);
// remove an entry
staff.remove("567-24-2546");
// replace an entry
staff.put("456-62-5527", new Employee("Mona"));
// look up a value
System.out.println(staff.get("157-62-7935"));
// iterate through all entries
staff.forEach((k, v) ->
System.out.println("key=" + k + ", value=" + v));
}
```

Map Example-Output

```
{157-62-7935=collections.maps.Employee@7960847b, 144-25-  
5464=collections.maps.Employee@6a6824be, 456-62-  
5527=collections.maps.Employee@5c8da962, 567-24-  
2546=collections.maps.Employee@512ddf17}  
collections.maps.Employee@7960847b  
key=157-62-7935, value=collections.maps.Employee@7960847b  
key=144-25-5464, value=collections.maps.Employee@6a6824be  
key=456-62-5527, value=collections.maps.Employee@5f5a92bb
```

Map Interface Methods

`V get(Object key);`

`default V getOrDefault(Object key, V defaultValue);`

`V put(K key, V value);`

`void putAll(Map<? extends K, ? extends V> entries);`

`boolean containsKey(Object key);`

`boolean containsValue(Object value);`

`default void forEach(BiConsumer<? super K, ? super V>
action);`

Updating Map Entries

- Normally, you get the old value associated with a key, update it, and put back the updated value.
- But you have to worry about the special case of the first occurrence of a key.
- Consider using a map for counting how often a word occurs in a file.
- When we see a word, we'd like to increment a counter like this:

`counts.put(word, counts.get(word) + 1);`

Updating Map Entries

- That works, except in the case when word is encountered for the first time. Then ***get*** returns null, and a ***NullPointerException*** occurs.
- A simple remedy is to use the ***getOrDefault*** method:
counts.put(word, counts.getOrDefault(word, 0) + 1);
- Another approach is to first call the ***putIfAbsent*** method. It only puts a value if the key was previously absent.

counts.putIfAbsent(word, 0);

counts.put(word, counts.get(word) + 1); // Now we know that get will succeed

Updating Map Entries

- But we can do better than that; we can use the *merge* method that simplifies this common operation as follows:

counts.merge(word, 1, Integer::sum);

- The Call to the *merge* method associates *word* with **1** if the key wasn't previously present, and otherwise combines the previous value and 1, using the *Integer::sum* function.

Map Views

- The collections framework does not consider a map itself as a collection.
- However, you can obtain ***views*** of the map—objects that implement the Collection interface or one of its sub interfaces.
- There are three views:
 - ***The Set of keys***
 - ***The Collection of values***
 - ***The Set of key/value pairs***

Map Views

- The following methods return these three views:
 - *Set<K> keySet()*
 - *Collection<V> values()*
 - *Set<Map.Entry<K, V>> entrySet()*

Algorithms

- Generic collection interfaces have a great advantage—you only need to implement your algorithms once.
- For example, consider a simple algorithm to compute the maximum element in a collection.
- Here is how you can find the largest element of an Array or ArrayList or LinkedList.

Algorithms

```
//Array
if (a.length == 0) throw new
NoSuchElementException();
T largest = a[0];
for (int i = 1; i < a.length; i++)
if (largest.compareTo(a[i]) < 0)
largest = a[i];
```

```
//ArrayList
if (v.size() == 0) throw new
NoSuchElementException();
T largest = v.get(0);
for (int i = 1; i < v.size(); i++)
if (largest.compareTo(v.get(i)) < 0)
largest = v.get(i);
```

```
//LinkedList
if (l.isEmpty()) throw new NoSuchElementException();
Iterator<T> iter = l.iterator();
T largest = iter.next();
while (iter.hasNext())
{
    T next = iter.next();
    if (largest.compareTo(next) < 0)
        largest = next;
}
```

Algorithms

- These loops are tedious to write, and just a bit error-prone.
 - Do the loops work correctly for empty containers?
 - For containers with only one element?
- You don't want to test and debug this code every time, but you also don't want to implement a whole set of methods for arrays, array lists, and linked lists.

Algorithms

- That's where the collection interfaces come in.
- As you have seen in the computation of the maximum element in a linked list, random access is not required for this task.
- Computing the maximum can be done simply by iterating through the elements. Therefore, you can implement the max method to take *any* object that implements the Collection interface.

Algorithms- Generic method for implementing Max

```
public static <T extends Comparable> T max(Collection<T> c)
{
    if (c.isEmpty()) throw new NoSuchElementException();
    Iterator<T> iter = c.iterator();
    T largest = iter.next();
    while (iter.hasNext())
    {
        T next = iter.next();
        if (largest.compareTo(next) < 0)
            largest = next;
    }
    return largest;
}
```

Now you can compute the maximum of a linked list, an array list, or an array, with a single method.

Algorithms- Sorting and Shuffling

- Sorting algorithms are part of the standard library for most programming languages, and the Java programming language is no exception.
- The ***sort*** method in the ***Collections*** class sorts a collection that implements the ***List*** interface.

```
List<String> staff = new LinkedList<>();  
fill collection  
Collections.sort(staff);
```

- This method assumes that the list elements implement the ***Comparable*** interface. If you want to sort the list in some other way, you can use the ***sort*** method of the ***List*** interface and pass a ***Comparator*** object.

Algorithms- Sorting and Shuffling

- Here is how you can sort a list of employees by salary:
staff.sort(Comparator.comparingDouble(Employee::getSalary));
- If you want to sort a list in *descending* order, use the static convenience method **Comparator.reverseOrder()**. For Example
staff.sort(Comparator.reverseOrder())
- sorts the elements in the list staff in reverse order, according to the ordering given by the compareTo method of the element type. Similarly,
staff.sort(Comparator.comparingDouble(Employee::getSalary).reversed())
sorts by descending salary.

Algorithms- Sorting and Shuffling

- The following example fills an array list with 49 Integer objects containing the numbers 1 through 49.
- It then randomly shuffles the list and selects the first six values from the shuffled list.
- Finally, it sorts the selected values and prints them.

Algorithms- Sorting and Shuffling example

```
import java.util.*;  
  
/**  
 * This program demonstrates the random shuffle and sort algorithms.  
 * @version 1.11 2012-01-26  
 * @author Cay Horstmann  
 */  
public class ShuffleTest{  
    public static void main(String[] args){  
        List<Integer> numbers = new ArrayList<>();  
        for (int i = 1; i <= 49; i++)  
            numbers.add(i);  
        Collections.shuffle(numbers);  
        List<Integer> winningCombination = numbers.subList(0, 6);  
        Collections.sort(winningCombination);  
        System.out.println(winningCombination);  
    }  
}
```

[9, 13, 22, 25, 27, 38]

Algorithms- Binary Search

- To find an object in an array, you normally visit all elements until you find a match.
- However, if the array is sorted, you can look at the middle element and check whether it is larger than the element that you are trying to find.
- If so, keep looking in the first half of the array; otherwise, look in the second half.
- That cuts the problem in half, and you keep going in the same way.

Algorithms- Binary Search

- For example, if the array has 1024 elements, you will locate the match (or confirm that there is none) after 10 steps.
- Whereas a linear search would have taken you an average of 512 steps if the element is present, and 1024 steps to confirm that it is not.
- The ***binarySearch*** method of the Collections class implements this algorithm.
- Note that the collection must already be sorted, or the algorithm will return the wrong answer.

Algorithms- Binary Search

- To find an element, supply the collection and the element to be located.
- If the collection is not sorted by the compareTo element of the Comparable interface, supply a comparator object as well.
i = Collections.binarySearch(c, element);
i = Collections.binarySearch(c, element, comparator);
- A non-negative return value from the binarySearch method denotes the index of the matching object.

Simple Algorithms

- The **Collections** class contains several simple but useful algorithms. Among them is the example from the beginning of this section—finding the maximum value of a collection.
- Others include copying elements from one list to another, filling a container with a constant value, and reversing a list.
- Java SE 8 adds default methods ***Collection.removeIf*** and ***List.replaceAll*** that are just a bit more complex. You provide a lambda expression to test or transform elements.

Simple Algorithms

- For example, here we remove all short words and change the remaining ones to lowercase:

```
words.removeIf(w -> w.length() <= 3);
```

```
words.replaceAll(String::toLowerCase);
```

- ***java.util.Collections*** contains a set of static methods that represents implementation of some of the most common operations and algorithms

Sample Methods from java.util.Collections

```

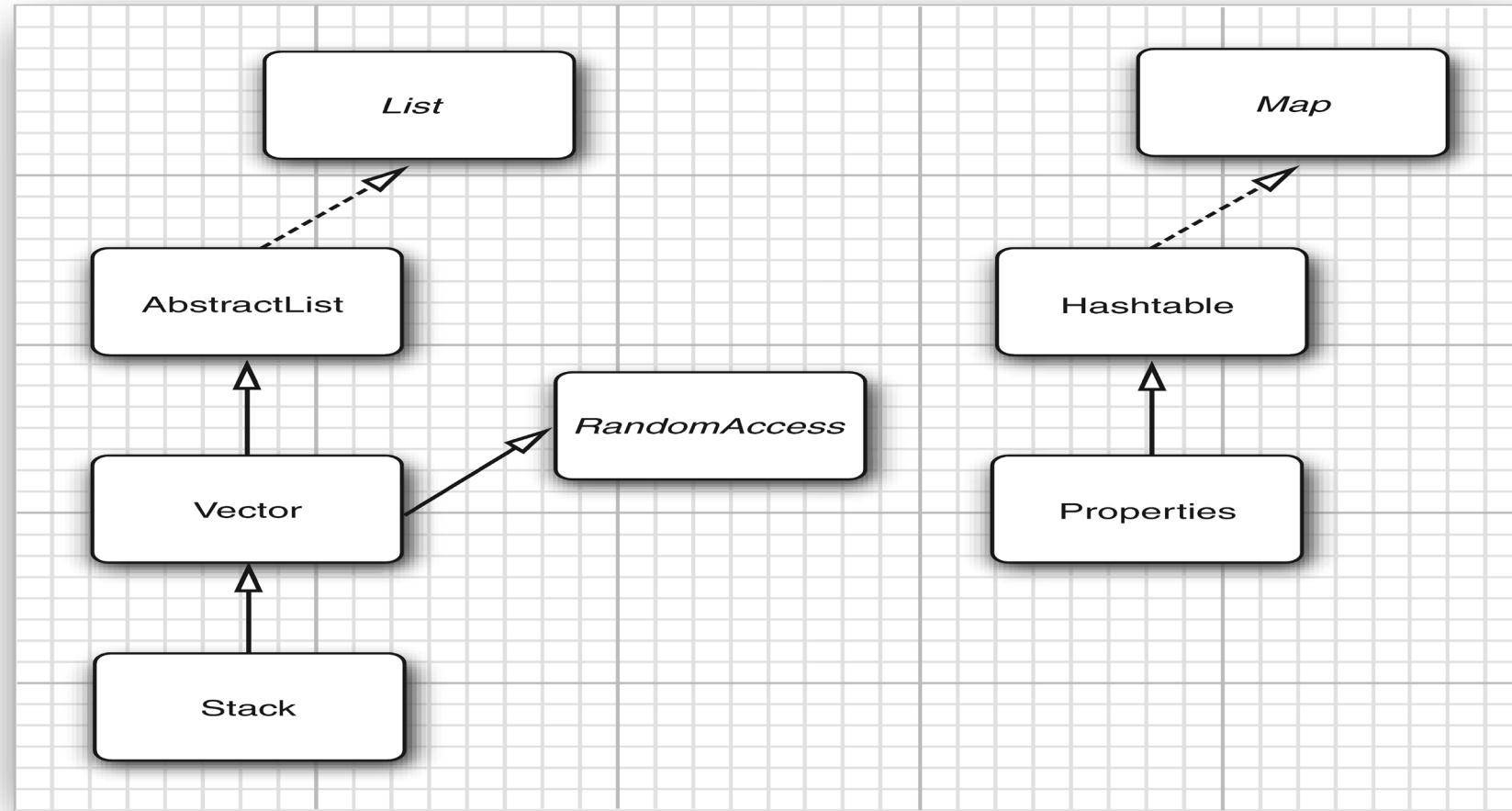
static <T> min(Collection<T> elements, Comparator<? super T> c)
static <T> max(Collection<T> elements, Comparator<? super T> c)
static <T> void copy(List<? super T> to, List<T> from)
static <T> void fill(List<? super T> l, T value)
static <T> boolean addAll(Collection<? super T> c, T... values)
static <T> boolean replaceAll(List<T> l, T oldValue, T newValue)
static void swap(List<?> l, int i, int j)
static void reverse(List<?> l)
static void rotate(List<?> l, int d)
static int frequency(Collection<?> c, Object o)
default boolean removeIf(Predicate<? super E> filter)
  
```

Rotates the elements in the specified list by the specified distance. After calling this method, the element at index i will be the element previously at index (i - distance) mod list.size()

Legacy Collections

- A number of “legacy” container classes have been present since the first release of Java, before there was a collections framework.
- They have been integrated into the collections framework as shown in the following image

Legacy Collections



Lab Exercise

- ***Simple Word Dictionary***
 - Develop an application that stores words in an array into a collection.
 - Create a map that uses the alphabets as keys and a collection as values (This collection should only contain words starting with the corresponding key)
 - Elements in the words map for each alphabet should be sorted
 - Provide methods to print all the letters and corresponding words
 - Provide a method to print the words of a given letter

A.	The History and Evolution of Java	2
B.	An Overview of Java	21
C.	Data Types, Variables and Arrays	64
D.	Operators, Control Statements and String Handling	98
E.	Packages and Interfaces	163
F.	Wrapper Classes, Enumeration, Autoboxing and Annotations	216
G.	Exception Handling	256
H.	Generics and Lambda Expressions	299
I.	Java Stream API	363
J.	Java Collections	430
K.	Java Platform Module System	522

Lesson 11

Java Platform Module System (JPMS)

Java Platform Module System (JPMS)

Introducing Java 9 Modularity

- We'll look at the impact of modularity on Java development and how you can use it to build powerful modular applications.
- We will be covering the following topics:
 1. Examining two important structural and organizational problems when building Java applications today, and their implications.
 2. Why does Java even need modularity features? What are we missing right now? And what do we gain from modularity?
 3. Introducing the **Java Platform Module System (JPMS)**
 4. Understanding the benefits that the Java modular system aims to provide

Modularity in Java

- The word **module** is perhaps one of the most overused terms in software development.
- A module can mean anything ranging from a group of code entities, components, or UI types, to framework elements to complete reusable libraries.
- When writing code, we typically try to break the code base down into smaller units in order to manage complexity.
- For anything more than very simple programs, having a monolithic code base is not a good idea.

Modularity in Java

- That's why *modular programming* is a generally favored software design approach.
- There are two important goals that modularity in software development usually achieves, which are as follows:
 1. *Divide and conquer approach*
 2. *Achieving encapsulation and well-defined interfaces*

Modularity in Java

- Starting with Java 9, Java developers now have the ability to create smaller units of code with a new construct called **Java modules** that they can group together like building blocks in order to compose larger applications.
- Java 9 also comes with what is probably the biggest renovation to the core Java code base itself.
- The **Java Runtime Environment (JRE)** and the **Java Development Kit (JDK)** have been rewritten to use the concepts of modularity so that the core Java Platform itself is modularized.

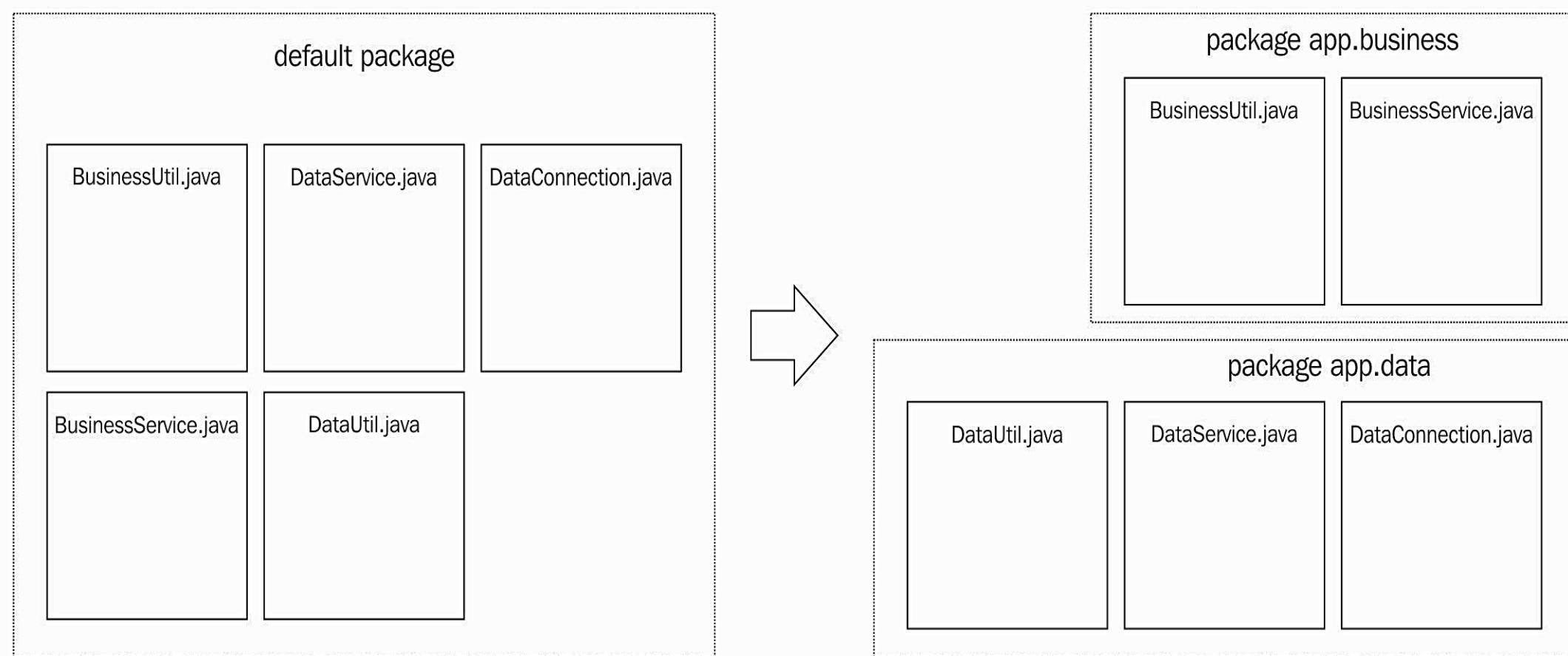
Modularity in Java

- Every Java class contains a portion of the overall application functionality that happens to belong together. We have the ability to *encapsulate* some functionalities as internal to a class (as private) and some others as external (or public).
- And then there's something in between with protected, thanks to the concept of packages.

Rethinking Java development with packages

- The idea of packages is to group your Java types into namespaces that signify the relationship, or perhaps a common *theme* among those types. It makes code easier to read, understand, and navigate.
- The following diagram shows an example of classes organized in packages. Adding all classes to a single package (left) is not good practice.
- We typically group related classes into well-named packages that describe the nature of the classes in them (right):

Rethinking Java development with packages



Rethinking Java development with packages

- The following table shows the various ways in which you can encapsulate code in Java before Java 9:

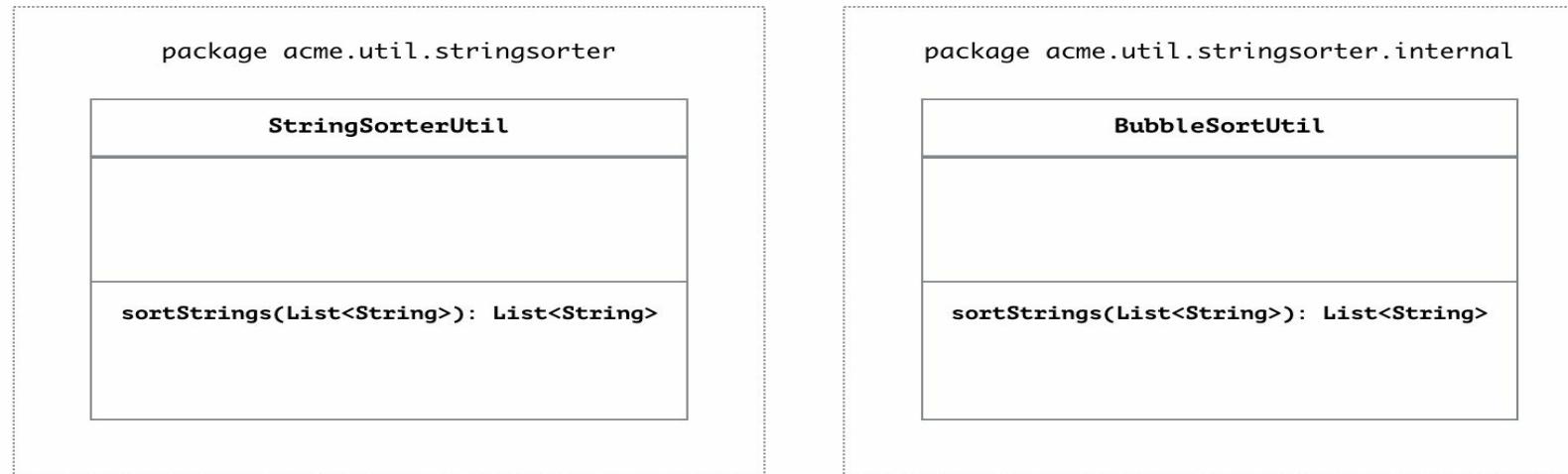
What to encapsulate	How to encapsulate	Encapsulation boundary
Member variables and methods	private modifier	Class
Member variables and methods	protected modifier	Package and subclasses
Member variables, methods, and types	No modifier (default package - protected)	Package

Rethinking Java development with packages

- The preceding table is where a limitation in the modular ability of the language becomes apparent.
- Most of the encapsulation features provided by these modifiers focus on controlling access to member variables and methods.
- The only way you can really protect access to a type is by making it package-protected.
- **There are a couple of problems with approaching modularity with just the preceding paradigm available in Java 8 and earlier.**

Rethinking Java development with packages

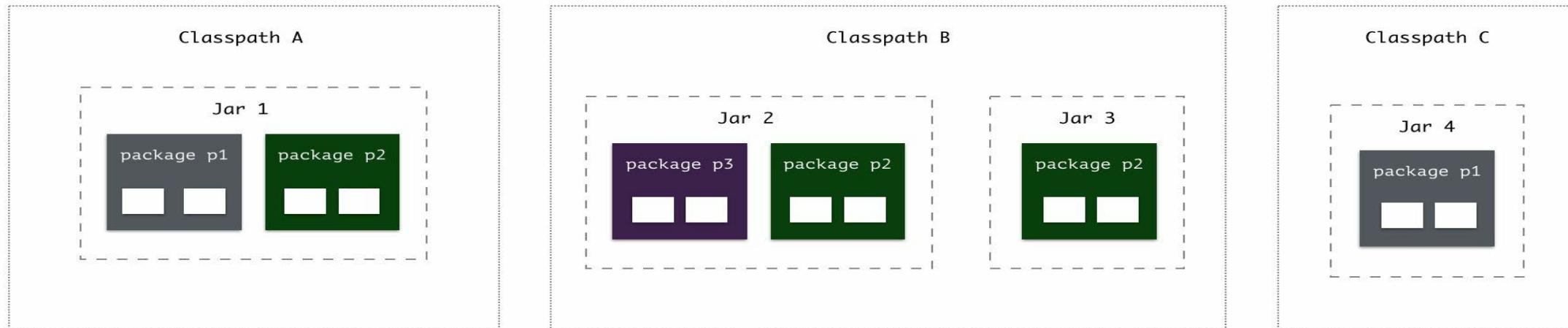
- Consider a library packaged in a jar file as follows:



- The main utility class was *StringSorterUtil* with one method-*sortStrings*. The method in turn internally called and delegated the sorting responsibility to the *BubbleSortUtil.sortStrings()* class from a class in the *acme.util.stringsorter.internal* package.

Rethinking Java development with packages

- **First problem:** how to protect the class in the internal library from not being used directly by library users?
- **Second problem:** Consider a deployment scenario that involves updating the following deployed application



Rethinking Java development with packages

- In Java, a classpath is just a set of paths. Any of those locations could have the jars and classes that the application needs to work.
- You can immediately see how easy it is for things to break! There's always a possibility that some of the classes that the application uses are *not* available in the classpath.
- If the runtime doesn't have a specific class it needs, the application could start running fine, but throw a ***NoClassDefFoundError*** much later.

Rethinking Java development with packages

- The two problems with packages and classpathes are:
 - Can we have an effective way to *encapsulate* portions of a library.
 - We needed a way to *ensure reliable execution* of an application without actually executing it.

Rethinking Java development with packages

- The classpath problem
 - A JAR file is just a convenient bundle of classes. Nothing more! Once in the classpath, the JVM treats classes in a JAR no differently from separate class files all in the same root directory.
 - What's worse is, once a class is in the classpath, It's incredibly easy for any developer to use a type they are not supposed to.
 - There's a problem commonly called **JAR hell**, which refers to several issues resulting from mismatched and incorrect classes and versions in JAR files.

Java Platform Module System

- The modularity features in Java 9 are together referred to by the name **Java Platform Module System (JPMS)**.
- It introduces a new language construct to create reusable components called ***modules***.
- The Java Platform Module System makes it easy for developers to create contained units or components that have clearly established dependencies on other modules

Java Platform Module System

- With Java 9 modules, you can group certain types and packages into a module and provide it with the following information:
 - Its name:** This is a unique name for the module
 - Its inputs:** What does the module need and use? What's required for the given module to be compiled and run?
 - Its outputs:** What does this module output or export out to other modules?

Java Platform Module System

- Whenever developers need to create any components that are meant to be reusable, they can create new Java modules and provide this information to create a unit of code with a clear interface.
- Since a module can contain both its inputs and outputs specified formally, it adds a whole set of advantages compared to the classpath approach that we've criticized so far.

Creating Your First Java Module

- 1. Create a module and give it a name:**
 1. Every module has a name associated with it, for the obvious purpose of referring to it.
 2. The recommended way to name a module is to use the *reverse domain name* convention, similar to the way you name your packages.
- 2. Define the module inputs:**
 1. Not many modules can realistically be self sufficient. You'll often need to import types that aren't in your module.
 2. You do that specifying which modules your module *requires*.

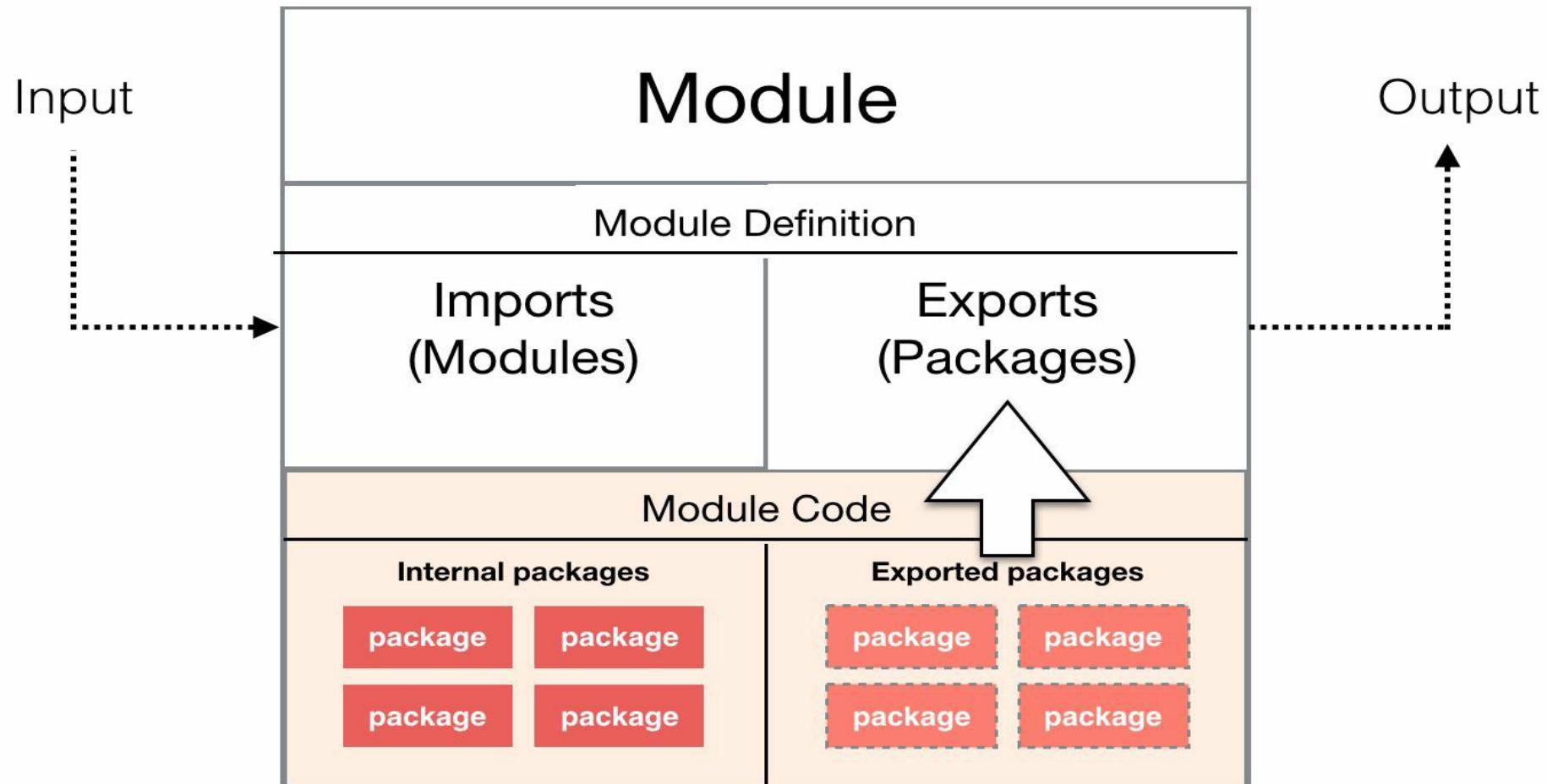
Creating Your First Java Module

3. Define the module outputs:

1. By default, every Java type you place in a module is accessible only to other types in the same module. Even if the type is marked public!
2. In order to expose types outside the module, you need to explicitly specify which *packages* you want to *export*.

- The following diagram illustrates the input and output definitions of a typical module:

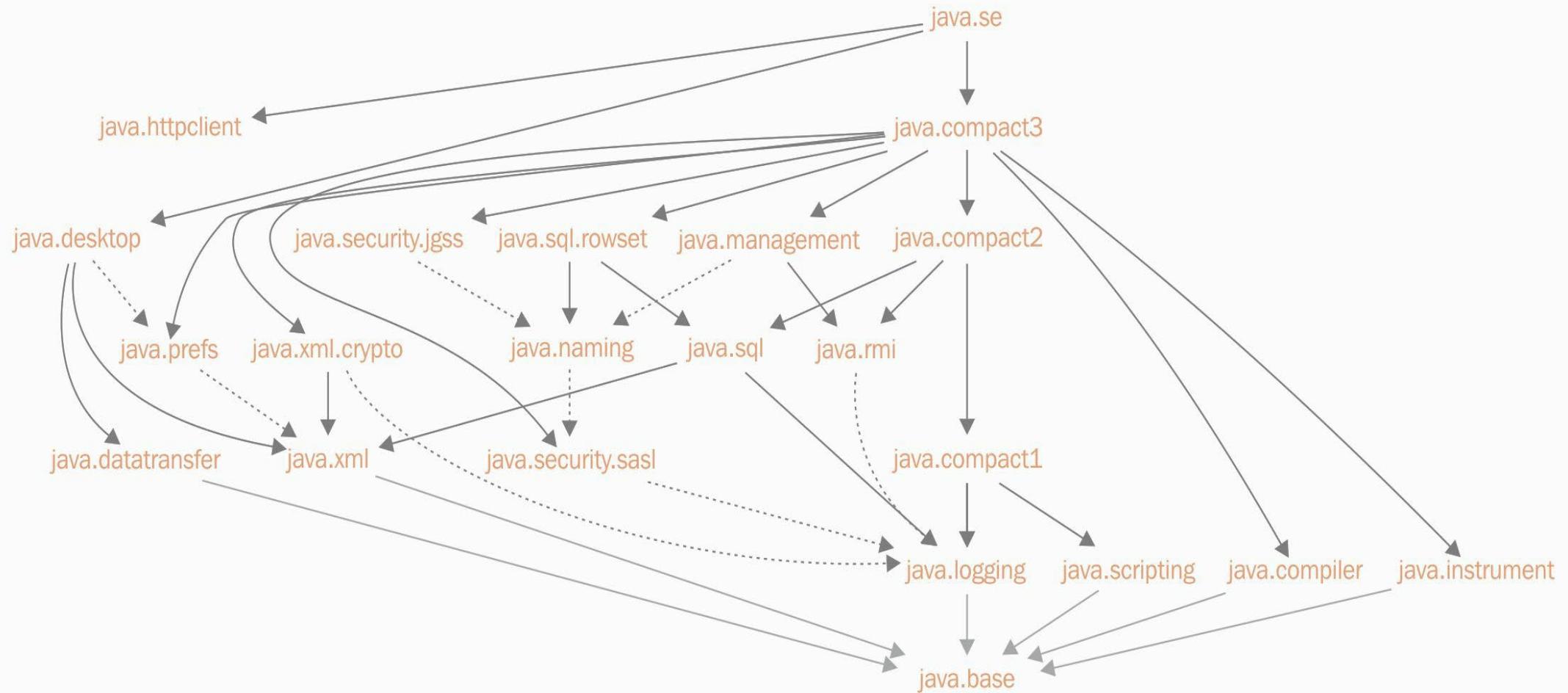
Creating Your First Java Module



Java Platform Module System

- JPMS was designed with two primary goals in mind:
 - **Strong encapsulation**
 - **Reliable configuration**
- In addition to these two core goals, there has been another important goal that the module system was designed for--to be scalable and easy to use even on huge monolithic libraries.
 - *Project Jigsaw: Modular development starts with a modular platform.*
- The following is an illustration of a subset of the new Java 9 platform modules.

Modular JDK

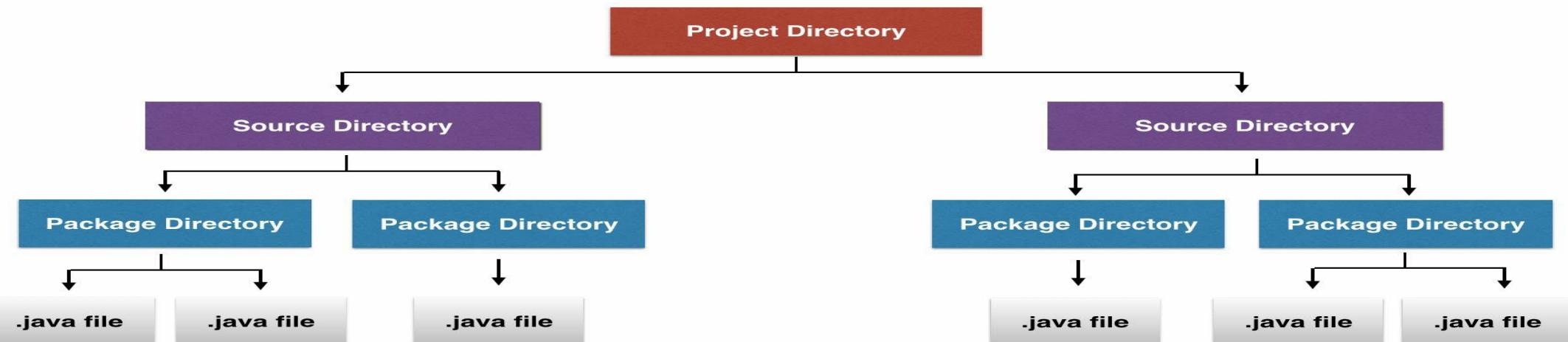


Java 9 modules

- When writing an application starting Java 9, you are ideally creating a modular application.
- A modular application calls for a completely new way of thinking about writing and structuring your code base.
- let's do a quick recap of the traditional Java code structure pre-Java 9.

Traditional Java code structure

- Steps to organize source code have typically been:
 - Create one or more source folders.
 - In a source folder, create package folders to mirror the package name.
 - Place the .java files in the right package folders:



What is a module?

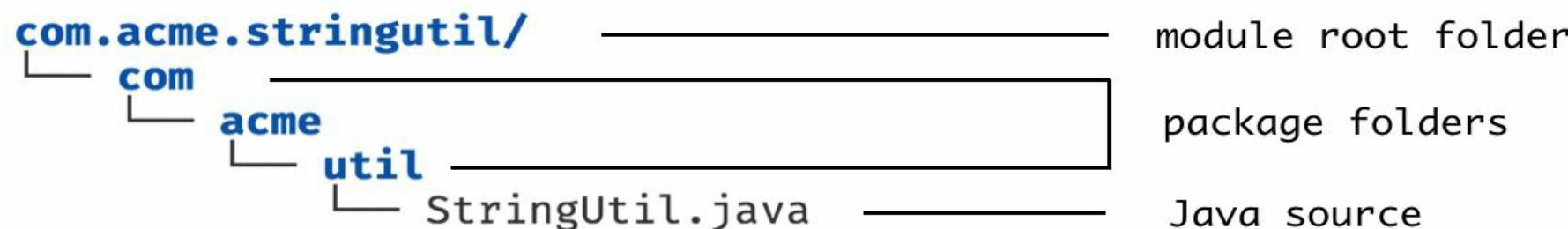
- The *module* is a new construct that has been introduced into the Java 9 programming language.
- A Java 9 module is a named, self describing collection of code and data that you can create and use in Java applications.
- A module can contain any number of Java packages that in turn contain Java code elements, such as classes or interfaces.
- A Java module can also contain files such as resource files or property files.
- The analogy of building blocks applies well here--a module is a building block that exists on its own but can be a part of a bigger whole.

Creating Your First Java Module

- Here are the top-level topics you'll be learning in this section:
 - Creating a new Java 9 module
 - Defining a module (using *module-info.java*)
 - Compiling and executing a module
 - Handling possible errors
- We will be building a sample Java 9 (or later) application. The application you'll build is an address book viewer application that displays some contacts sorted by last name

Creating a module

- 1. Assign a module name**
- 2. Create a module root folder:** The module root folder has the same name as the module.
- 3. Add the module code:** Inside the module root folder goes the code that belongs to the module. This begins with packages, so you start your package folders from the module root folder onward.

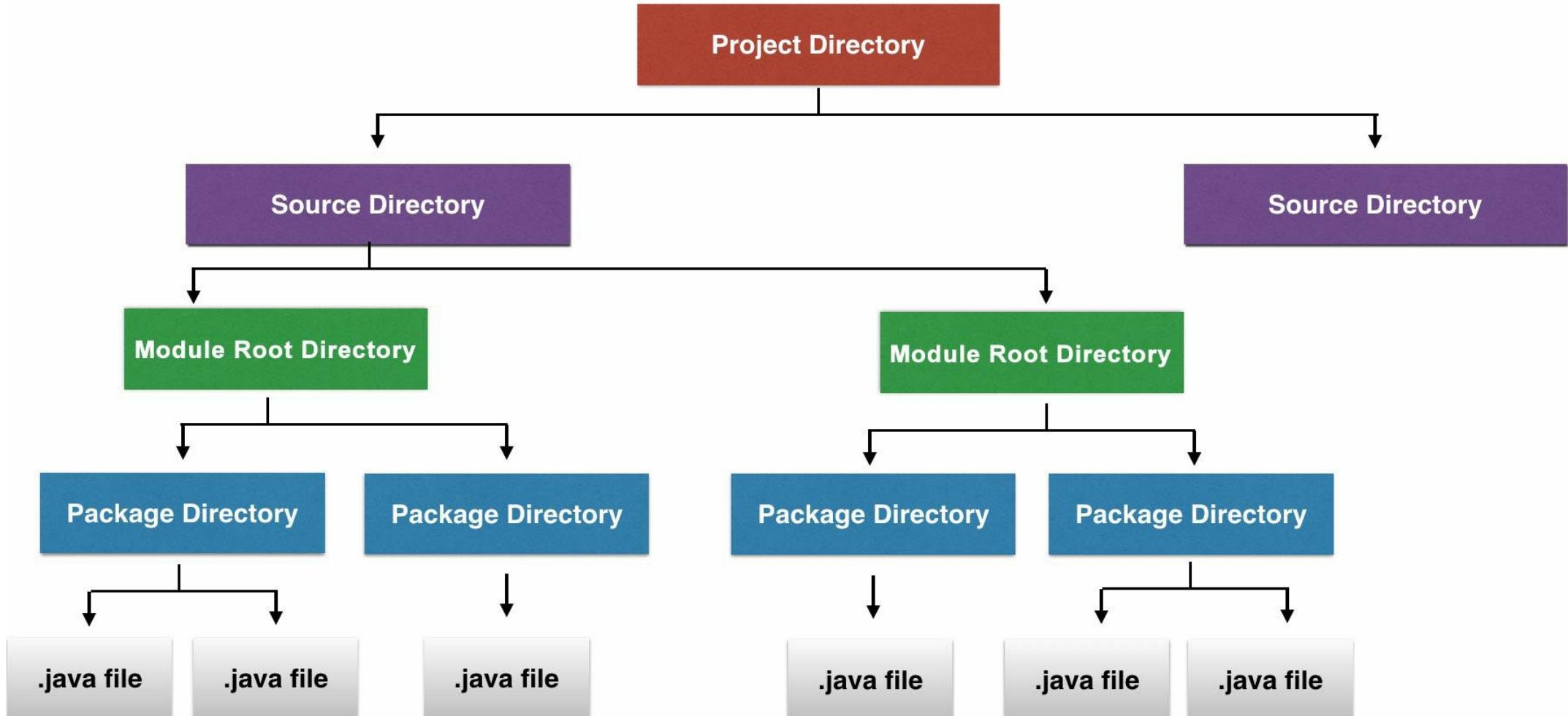


Creating a module

4. Create and configure the module descriptor

- Here's the final step! Every module comes with a file that describes it and contains metadata about the module.
- This file is called the module descriptor. This file contains information about the module, such as what it requires (the inputs to the module) and what the module exports (the outputs from the module).
- The module descriptor is always located directly at the module root folder, and it is always given the name `module-info.java`.

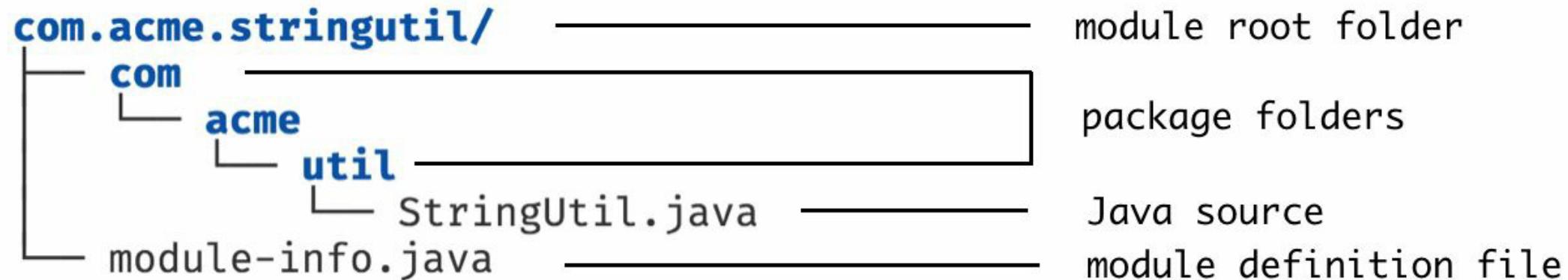
Creating a module



module-info.java

- Here is a minimal module descriptor for the example module:

```
module com.acme.stringutil {  
}
```



Implementing our first module

- 1. Name the module:** The name of our address book module is *packt.addressbook*.
- 2. Create a module root folder:** You'll now need to create one module folder for each module you intend to write. Since we are creating only one module called packt.addressbook, we create a folder with the same name.

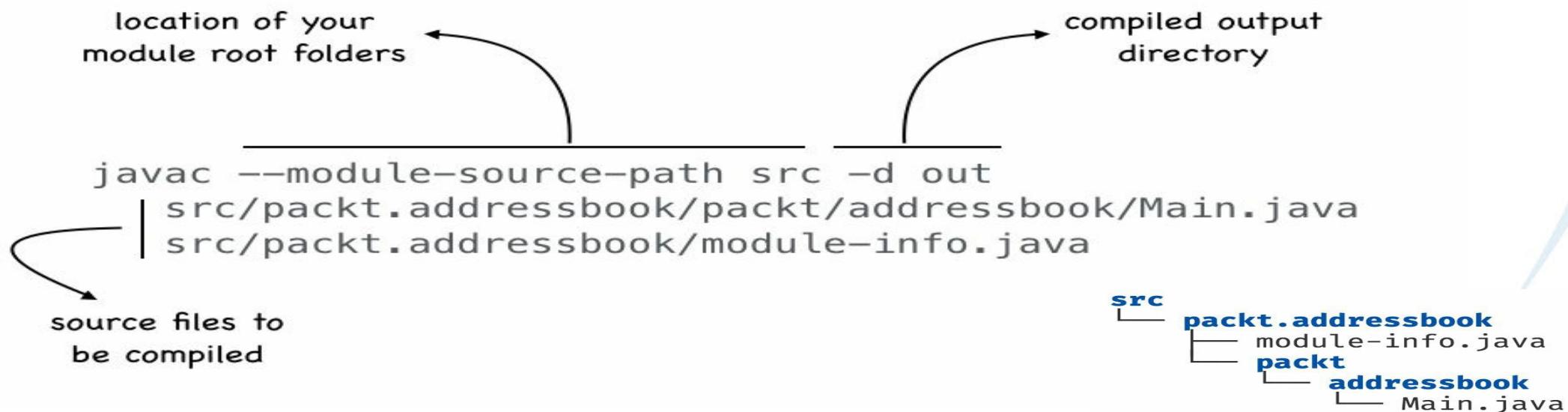


Compiling our module

- To compile modules in Java 9, you need to provide the javac command with the following information:
 - The location of your modules. ***This is the directory that contains the module root folders for all the modules in your application.*** In our case, this is the `src` folder.
 - The paths and names of the Java files that need to be compiled.
 - The destination location where the compiler needs to output the compiled `.class` files. This can be any location, but it is recommended choosing a folder named ***out*** directly below the project root folder.

Compiling our module

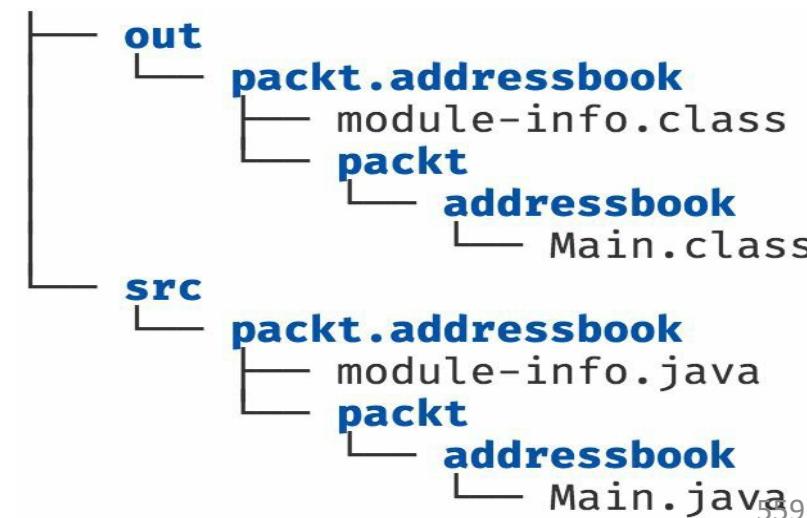
- To compile the module you've written, go to the project root (in our example, it's */code/java9*) and run the following command:



Compiling our module

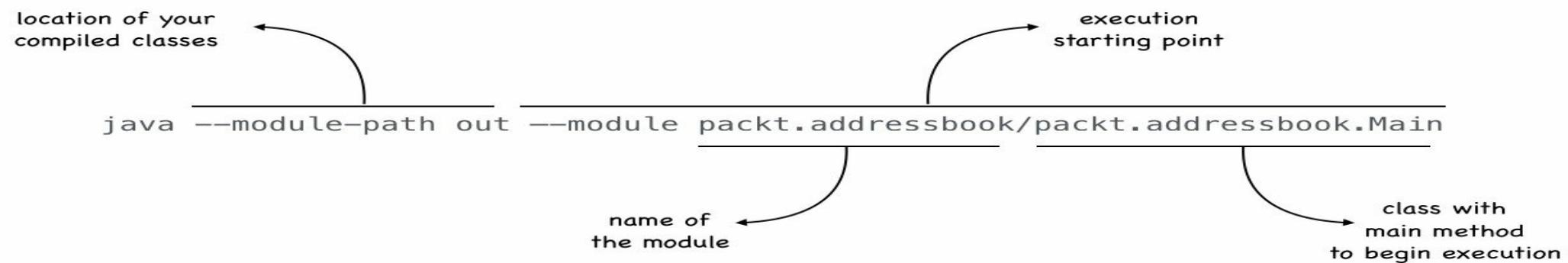
- Here, you are specifying the module source path
- (1) the module source path using ***the –module-source-path*** command option
- (2) the output directory for compiled classes using the **-d** command option
- (3) and the list of Java source files (3) by specifying them directly in the command (in this case, Main.java and moduleinfo.java).

If the compiler is successful, there is no output to the console. The out directory should contain the compiled classes:



Executing our module

- Here is the information you need to tell the java command in this case:
 - The location of the compiled modules--also called the **module path**.
 - The module that contains the class with the main method that needs to start the execution.
 - The class with the main method in the preceding module that needs to start the execution.



Another Module Example

- As an example, we are going to use a class with a method that returns a random, predefined quote related to programming:

```
package com.example.programming;

import java.util.Random;

public class ProgrammingQuotes {
    private String[] quotes = new String[]
    { "\"To iterate is human, to recurse divine.\n\""
    + "- L. Peter Deutsch",
    "\"Don't worry if it doesn't work right. \""
    + "If everything did, you'd be out of a job.\n"
    + "- Mosher's Law of Software Engineering",
    "\"Good design adds value faster than it adds cost.\n\""
    + "- Thomas C. Gale",
    "\"Talk is cheap. Show me the code.\n\""
    + "- Linus Torvalds",
    "\"I don't care if it works on your machine! We are not shipping your machine!\n\""
    + "- Vidiu Platon", };
```

Another Module Example

```
private Random rand = new Random();
private int getRandomIndex() {
    return rand.nextInt(quotes.length);
}
public String getQuote() {
    return quotes[getRandomIndex()];
}
public static void main(String args[]) {
    System.out.println(new ProgrammingQuotes().getQuote());
}
```

Another Module Example

- The directory structure of our project should be something like this:

com.example.programming ---Module Root Folder

module-info.java ---Module descriptor file

com

example

programming

ProgrammingQuotes.java

Another Module Example

- The module descriptor of our project should be as follows:

```
module com.example.programming {  
}
```

- Now open a terminal window, make sure to cd into the base directory, and compile as usual:**

```
javac -d out module-info.java com/example/programming/ProgrammingQuotes.java
```

- The directory out will be created with the compiled .class files for module-info.java and ProgrammingQuotes.java.

Another Module Example

- Next, package the class into a modular JAR:

```
jar cvfe programming-quotes.jar com.example.programming.ProgrammingQuotes -C out .
```

- Now run the jar file with the following command

```
java –jar programming-quotes.jar
```

c create the archive

v verbose

f file

e main class

C Change to the specified directory and include the following file

Another Module Example

- To run this program as a module, you have to specify the module path (which contains modules, unlike the classpath that contains classes) with the option --module-path (or just -p) and the main module/class with the option --module (or just -m), in the format module/class:
- Now run the jar file with the following command

```
java --module-path programming-quote.jar --module  
com.example.programming/com.example.programming.ProgrammingQuotes
```

- Or use the *out* directory that contains the compiled class

```
java --module-path out --module  
com.example.programming/com.example.programming.ProgrammingQuotes
```

Another Module Example

- In the following slides we will enhance the example by adding GUI to our application and learn how to use and export modules.
- Later we will learn how to use ServiceLoader with modules

Another Module Example

- Create another top-level directory com.example.gui with the following class:

```
package com.example.gui;

import javafx.application.Application;
import javafx.scene.*;
import javafx.stage.Stage;
import javafx.scene.control.*;

public class QuoteFxApp extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("Quotes");
    }
}
```

Another Module Example

```
Label label = new Label("A quote");
    Scene scene = new Scene(label, 500, 200);
    primaryStage.setScene(scene);
    primaryStage.show();
}
public static void main(String[] args) {
    Application.launch(args);
}
javac --module-source-path src -d out
src/com/example/gui/com/example/gui/QuoteFXApp.java
src/com/example/gui/module-info.java
```

Another Module Example

- For the code to compile successfully we need to require JavaFX Modules in our *module-info.java*

```
module com.example.gui {  
    requires javafx.controls;  
}
```

If we run the program with

```
java --module-path out --module  
com.example.gui/com.example.gui.QuoteFxApp
```

We'll get the following error:

```
java.lang.IllegalAccessException: class com.sun.javafx.application.LauncherImpl (in module  
javafx.graphics) cannot access class com.example.gui.QuoteFxApp (in module com.example.gui)  
because module com.example.gui does not export com.example.gui to module javafx.graphics
```

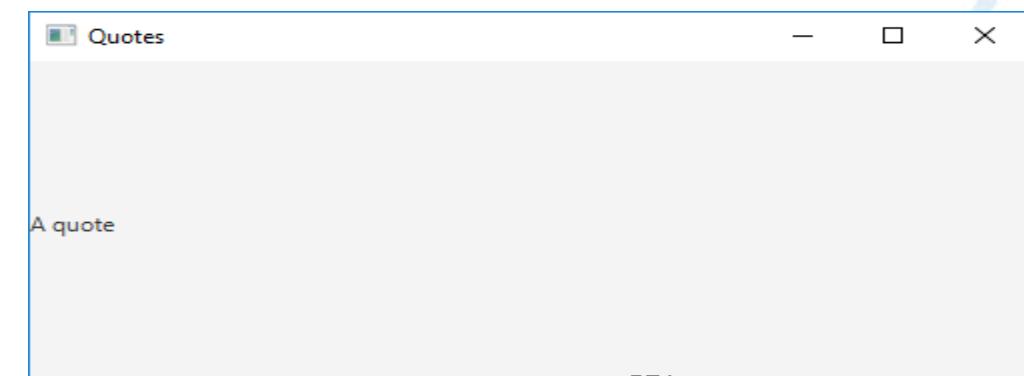
Another Module Example

- JavaFX needs access to our class *QuoteFxApp*, so we need to grant access explicitly by exporting the package *com.example.gui* this way:

exports com.example.gui; or use a qualified export

exports com.example.gui to javafx.graphics;

Our application should look like that:



Another Module Example

- Having learned how to require/export other modules, integrating the programming quotes module and the GUI module should be easy.
- *First, in the module com.example.programming, export its package:*

```
module com.example.programming {  
    exports com.example.programming;  
}
```

- Compile the classes and package them as a JAR file once again. We can copy this JAR to a lib directory for easy access.

Another Module Example

- Next, modify the class QuoteFxApp to use ProgrammingQuotes:

```
Label label = new Label(new  
ProgrammingQuotes().getQuote());
```

- Now we need to require the com.example.programming module:

```
module com.example.gui {  
    requires javafx.controls;  
    requires com.example.programming;  
}
```

Another Module Example

- Compile the ***com.example.gui*** module with the module path option so it can locate the ***com.example.programming*** module (which in this case, it's placed in the lib directory):

```
javac -d out --module-path ../lib module-info.java  
com/example/gui/QuoteFxApp.java
```

- Finally, run the application using, once again, the module path (remember, the path separator character for Windows is ;. Use : for Linux/Mac):

```
java --module-path out;../lib --module  
com.example.gui/com.example.gui.QuoteFxApp
```

Using Services with the ServiceLoader

- Let's say that now we want to show math quotes, in addition to programming quotes. We could create another module for math quotes and modify the GUI module to require it.
- We can also create an interface for both types of quotes. In the Java Platform Module System, this will give us the possibility to abstract the mechanism for matching up service interfaces with implementations using the service locator pattern.
- This works by using the service-provider loading facility that Java provides with the ServiceLoader class.

Using Services with the ServiceLoader

- To use this mechanism, you need an interface, abstract class, or even a concrete class to act as the type of service, an implementation or subclass, and use the class ***ServiceLoader*** to load all the implementations found.
- In the JPMS, this is done in the module-info.java file. Let's see it in action.

Using Services with the ServiceLoader

```
package com.example.quote;  
public interface Quote {  
    String getQuote();  
}
```

Export the interface's package in the module-info.java file:

```
module com.example.quote {  
    exports com.example.quote;  
}
```

Using Services with the ServiceLoader

Compile and package as usual

```
# Compilation  
javac -d out module-info.java com/example/quote/Quote.java
```

```
# Packaging (placing the jar in the common lib directory)  
jar cvf ..//lib/quote.jar -C out .
```

- Now modify the ProgrammingQuotes class to implement the interface.
- Don't forget to add the interface module to the module-info.java file of ***com.example.programming*** module:

Using Services with the ServiceLoader

- And now, instead of exporting the package of the module, with the help of the keyword ***provides*** you can indicate that this module provides an implementation of the Quote interface:

```
module com.example.programming {  
    requires com.example.quote;  
  
    provides com.example.quote.Quote  
        with com.example.programming.ProgrammingQuotes;  
}
```

Using Services with the ServiceLoader

- Next, compile and package:

```
# Compilation  
javac -d out --module-path ..//lib module-info.java  
com/example/programming/ProgrammingService.java  
  
# Packaging (placing the jar file in a common lib directory)  
jar cvf ..//lib/programming-service.jar -C out .
```

- Notice that we're not exporting the package of this module. We're just defining the interface and its implementation.
- The implementation is encapsulated in a non-exported package. This is generally the case because the point is to hide implementation details.

Using Services with the ServiceLoader

- In our example we can consider the GUI as the service consumer, and this is where we use the ServiceLoader class like this :

Using Services with the ServiceLoader

```
import com.example.quote.Quote;
import java.util.ServiceLoader;
public class QuoteFxApp extends Application {
    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("Quotes");
        Label label = new Label("NO QUOTE");
        ServiceLoader.load(Quote.class)
            .forEach(service ->
label.setText(service.getQuote()));
        Scene scene = new Scene(label, 500, 200);
        ...
    }
    ...
}
```

Using Services with the ServiceLoader

- *ServiceLoader.load(Class<T>)* returns a ServiceLoader instance that implements **Iterable**. So, we can iterate over all the discovered implementations of the provided interface.
- Next, in the module-info.java file, you just have to indicate that the module will require the module that contains the interface and it will use its implementations:

```
module com.example.gui{  
    ...  
    requires com.example.quote;  
    uses com.example.quote.Quote;  
}
```

Using Services with the ServiceLoader

- *Compile and run like before:*

```
# Compile
javac -d out --module-path ../lib module-info.java
com/example/gui/QuoteFxApp.java
```

```
# Run (remember to change the path separator if needed)
java --module-path out;../lib --module
com.example.gui/com.example.gui.QuoteFxApp
```

Creating a module using NetBeans

New Project

Steps

1. Choose Project
2. ...

Choose Project

Categories:

- Java
- JavaFX
- Maven
- NetBeans Modules
- Samples

Projects:

- Java Application
- Java Class Library
- Java Project with Existing Sources
- Java Modular Project**
- Java Free-Form Project

Description:

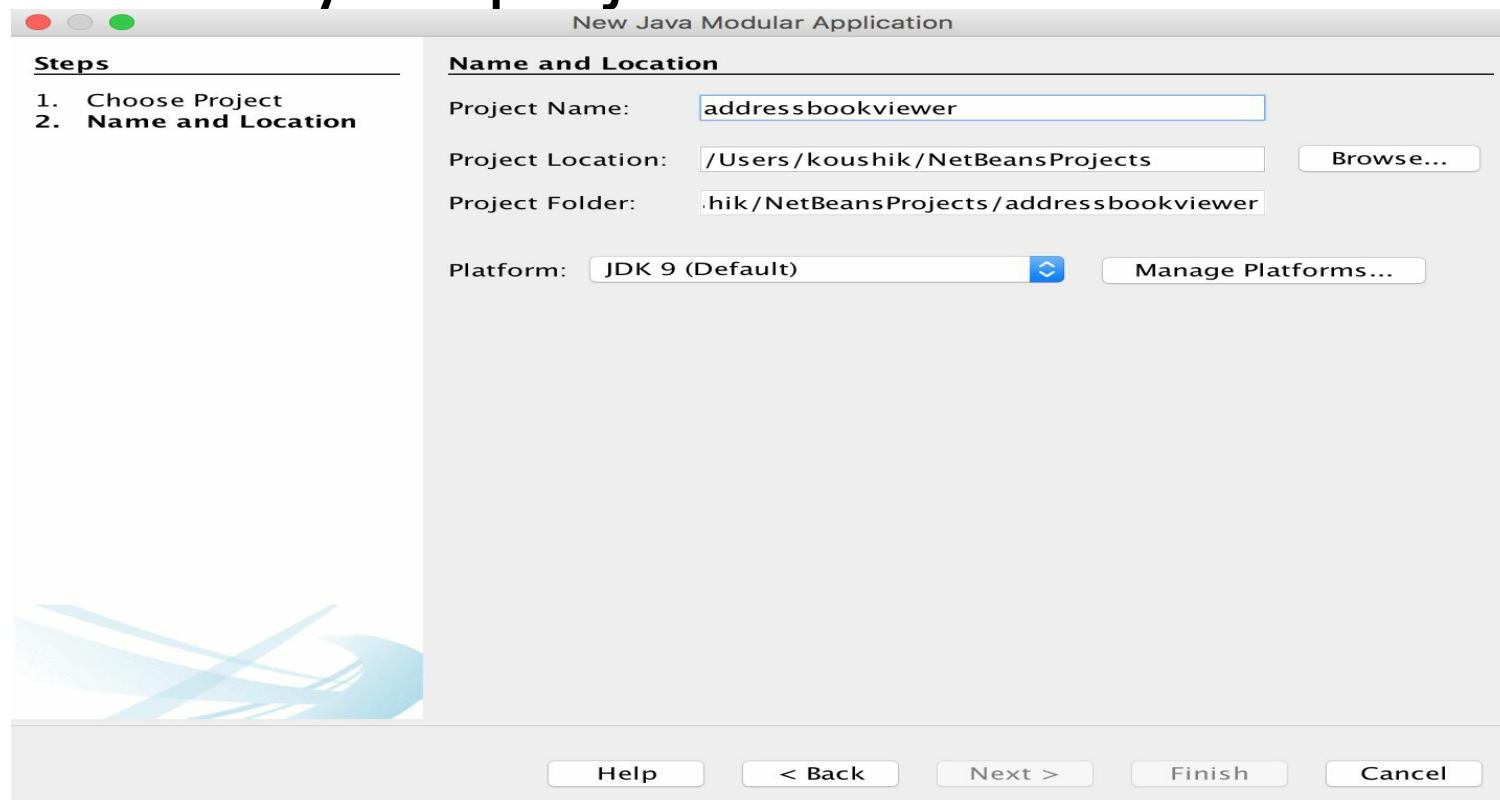
Creates a new Java SE modular application in a standard IDE project.
Multiple modules (as specified in JDK 9) can be added to your project
Standard projects use an **IDE-generated Ant build script** to build, run,
and debug your project.

Help < Back **Next >** Finish 585 Cancel

3-Dec-23

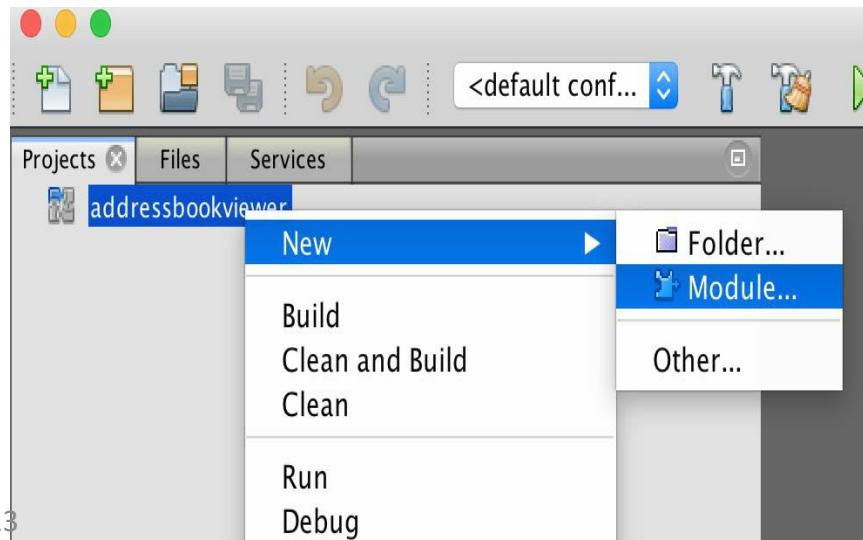
Creating a module using NetBeans

2) Select that and click Next. In the next dialog, you can specify the name of your project (I choose addressbookviewer) and the location of your project and click Finish.

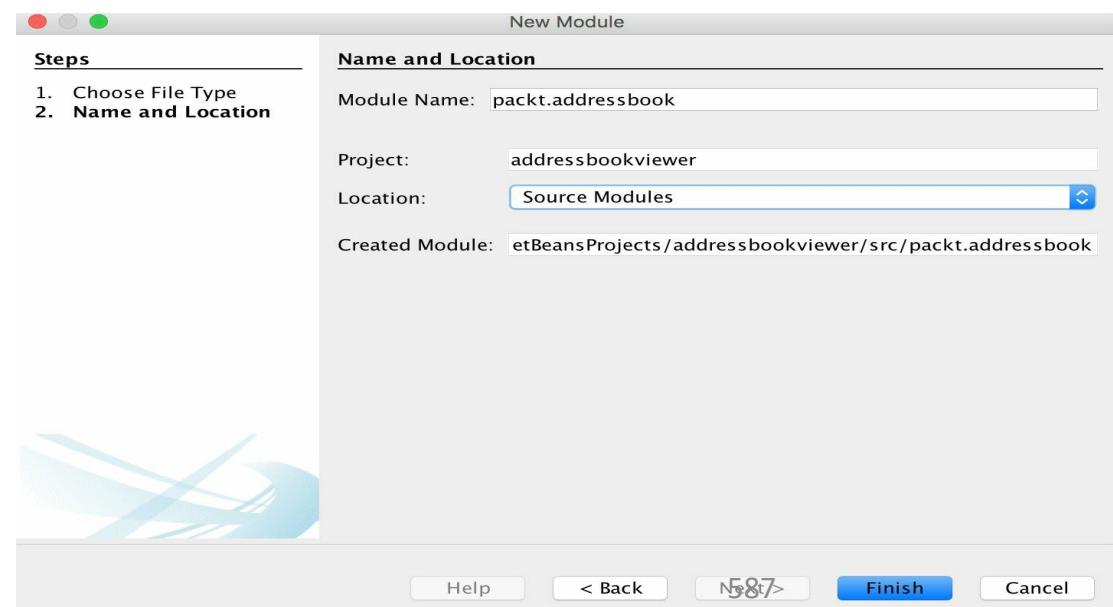


Creating a module using NetBeans

3) Once the new project is loaded onto your IDE, you can right-click on the name of the project in the Projects tab and choose the option to create a new module



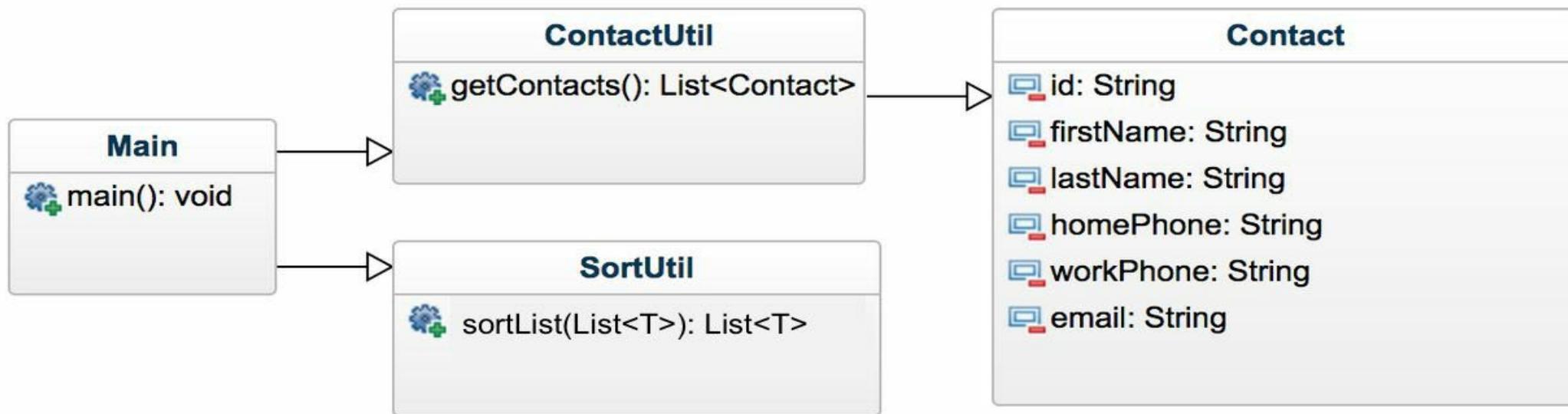
4) In the New Module dialog, enter the name of the module ***packt.addressbook*** and click Finish



Lab Exercise

The address book viewer application

- The following informal class diagram shows how we'll design the application classes to begin with:



The address book viewer application

