

Java Handout

- Introduction to Java
- Java Language Fundamentals
- Object Oriented Programming
- Arrays in Java
- Introduction to Java API
- Exception Handling
- Collection Framework
- Lambda Expression and Stream API
- Java-8 Date/Time API
- IO Streams, Loggers
- Java Database Connectivity



Introduction to Java

Introduction to Java

James Gosling

- [James Gosling](#) is generally credited as the inventor of the Java programming language
- He was the first designer of Java and implemented its original compiler and virtual machine
- He is also known as the Father of Java



What is Java



A multi-platform, network-centric, object-oriented programming language

- Multi-platform
 - ✓ It can run on almost any computer platform. It is platform independent.
- Network-centric
 - ✓ Designed with network in mind – “the network is the computer”
 - ✓ Designed for building applications for the Internet
- Object-oriented
 - ✓ It incorporates object-oriented programming model
- Write Once, Run Anywhere
 - ✓ Java is portable and platform independent
- Robust
 - ✓ Strong type checking
 - ✓ Exception handling mechanism
 - ✓ Automatic memory management
- Security
 - ✓ Can download remote code over a network and run it in a secure environment
 - ✓ Security levels and restrictions are highly configurable

Java Editions

A Java Platform is the set of APIs, class libraries, and other programs used in developing Java programs for specific applications

Java Platform Editions

Java Platform, Standard Edition (JSE)

- Core Java Platform targeting applications running on workstations

Java Platform, Enterprise Edition (JEE)

- Component-based approach to developing distributed, multi-tier Enterprise applications

Java SE Components

Java SE Includes Java Development Kit (JDK) and Java Runtime Environment (JRE)

Java Development Kit (JDK)

- Is a set of Java tools for developing Java programs
- Consists of Java API, Java Compiler, Debugger and JVM

Java Runtime Environment (JRE)

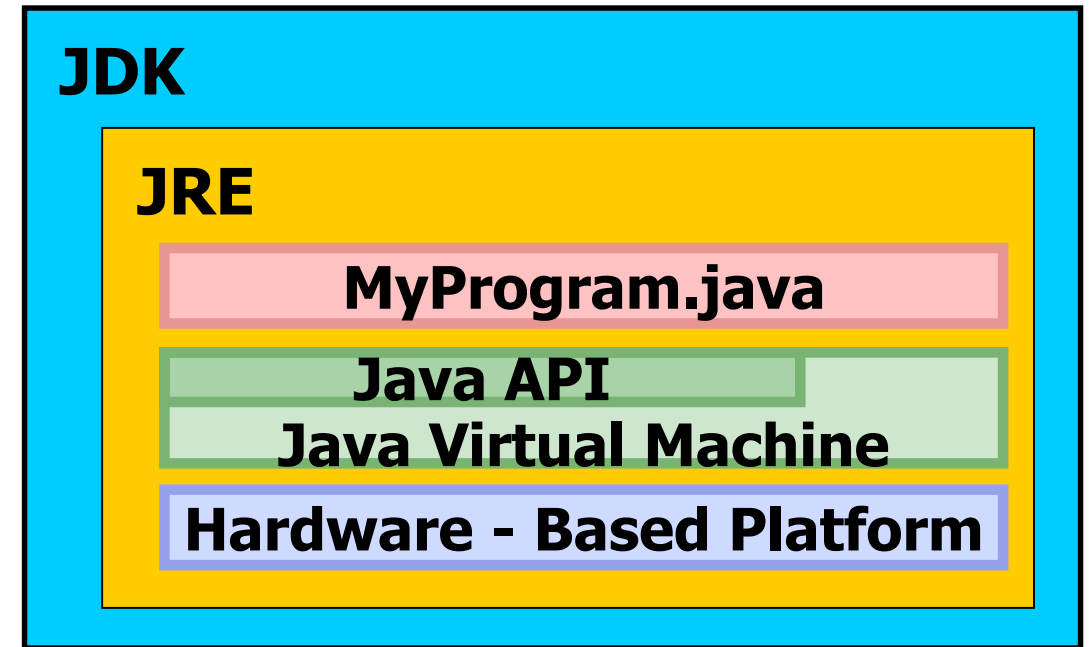
- A JRE provides JVM, standard class libraries(API) and other components to run applications written in Java

Java Application Programming Interface (API)

- Is prewritten code, organized into packages of similar topics

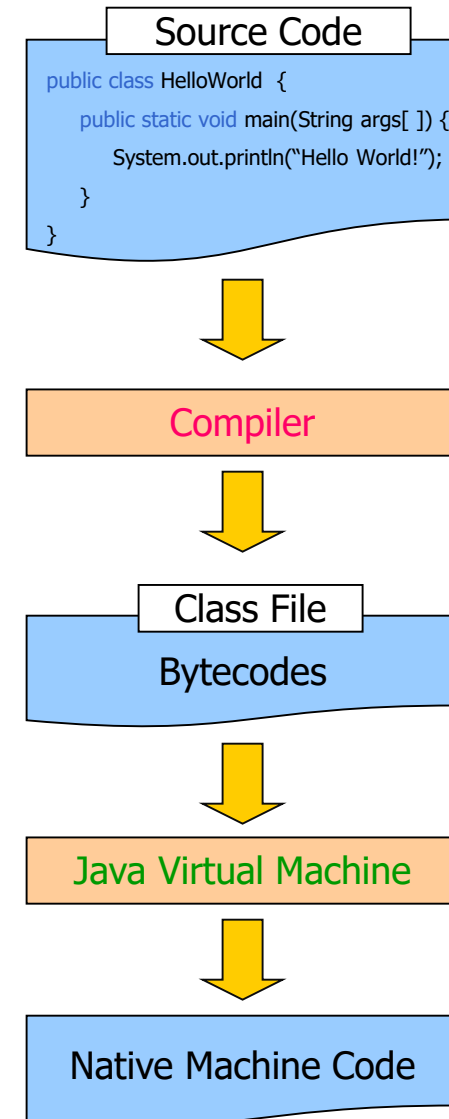
Java Virtual Machine (JVM)

- Is an execution engine that runs compiled Java byte code

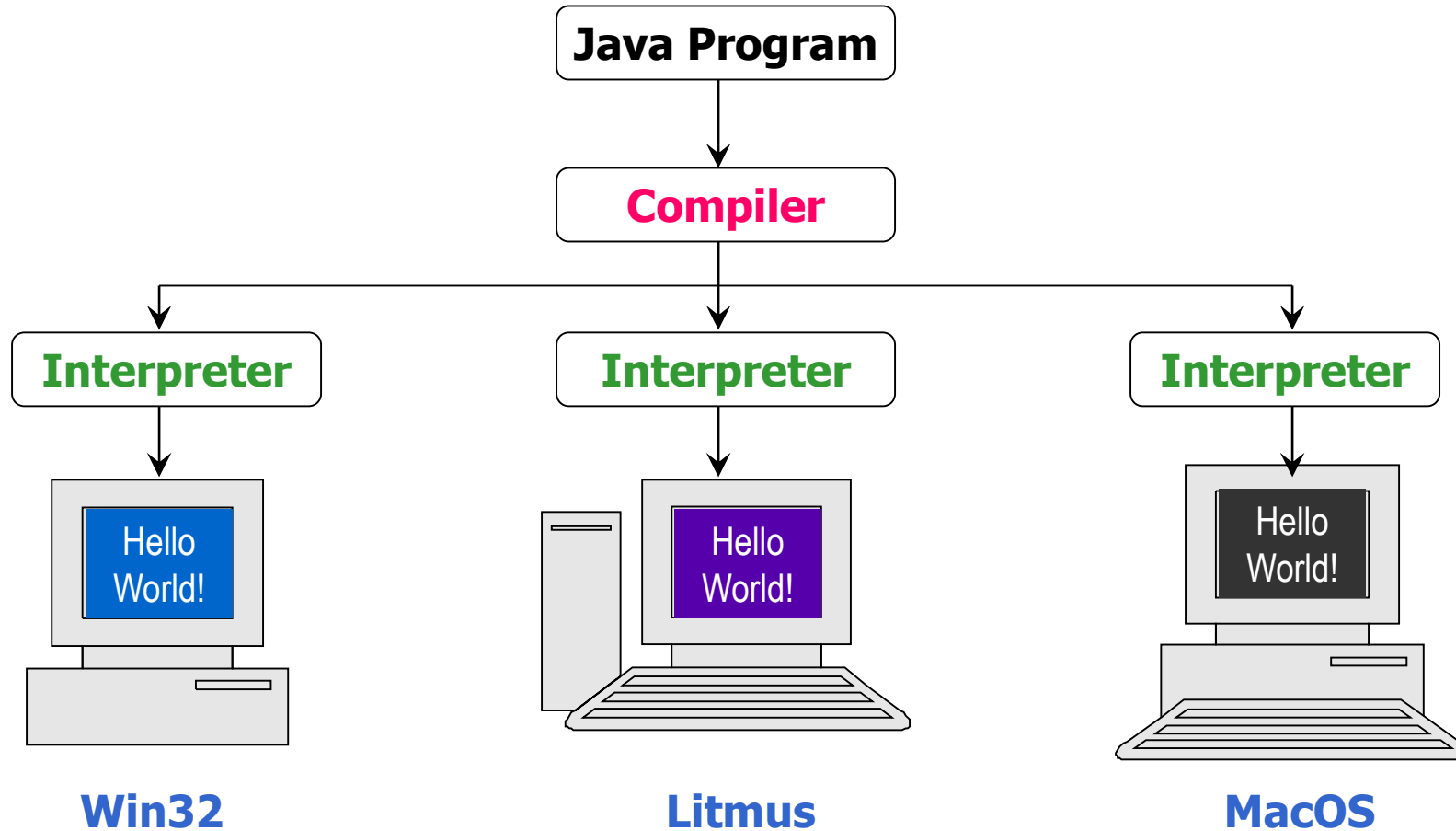


How does Code execution

- A Java program is written
- The program is compiled
- A *class file* is produced containing *bytecodes*
- The bytecodes are interpreted by the JVM
- The JVM translates bytecodes into native machine code



Platform independent



Java Language Fundamentals

Java Language Fundamentals

- Java Source File Structure
- Java Variables and Datatype
- Casting
- Operators
- Flow Controls

Java Source File Structure

1. Package declaration

Used to organize a collection of related classes.

2. Import statement

Used to reference classes and declared in other packages.

3. Class declaration

A Java source file can have several classes but only one public class is allowed.

```
/*
 * Created on Jun 09, 2022
 *
 * First Java Program
 */
package com.ibsplc.sample;
import java.util.*;

/**
 * @author IBS L&D
 */
public class JavaMain {

    public static void main(String[] args) {
        // print a message
        System.out.println("Welcome to Java!");
    }
}

class Extra {
    /*
     * class body
     */
}
```

Java Source File Structure

Comments

1. Single Line Comment

```
// insert comments here
```

2. Block Comment

```
/*  
 * insert comments here  
 */
```

3. Documentation Comment

```
/**  
 * insert documentation  
 */
```

Whitespaces

Tabs and spaces are ignored by the compiler. Used to improve readability of code.

```
/*  
 * Created on June 09, 2022  
 *  
 * First Java Program  
 */  
package com.ibsplc.sample;  
import java.util.*;  
  
/**  
 * @author IBS L&D  
 */  
public class JavaMain {  
  
    public static void main(String[] args) {  
        // print a message  
        System.out.println("Welcome to Java!");  
    }  
}  
  
class Extra {  
    /*  
     * class body  
     */  
}
```

Java Source File Structure

Class

- Every java program includes at least one class definition. The class is the fundamental component of all Java programs.
- A class definition contains all the variables and methods that make the program work. This is contained in the class body indicated by the opening and closing braces.

```
/*
 * Created on Jun 09, 2022
 *
 * First Java Program
 */
package com.ibsplc.sample;
import java.util.*;

/**
 * @author IBS L&D
 */
public class JavaMain {

    public static void main(String[] args) {
        // print a message
        System.out.println("Welcome to Java!");
    }
}

class Extra {
    /*
     * class body
     */
}
```

Java Source File Structure

main() method

This line begins the `main()` method. This is the line at which the program will begin executing.

String args[]

Declares a parameter named `args`, which is an array of `String`. It represents command-line arguments.

```
/*
 * Created on Jun 09, 2022
 *
 * First Java Program
 */
package com.ibsplc.sample;
import java.util.*;

/**
 * @author IBS L&D
 */
public class JavaMain {

    public static void main(String[] args) {
        // print a message
        System.out.println("Welcome to
Java!");
    }

}

class Extra {
    /*
     * class body
     */
}
```

Java Source File Structure

Java statement

- A complete unit of work in a Java program.
- A statement is always terminated with a semicolon and may span multiple lines in your source code.

System.out.println()

This line outputs the string "Welcome to Java!" followed by a new line on the screen.

```
/*
 * Created on Jun 09, 2022
 *
 * First Java Program
 */
package com.ibsplc.sample;
import java.util.*;

/**
 * @author IBS L&D
 */
public class JavaMain {

    public static void main(String[] args) {
        // print a message
        System.out.println("Welcome to Java!");
    }
}

class Extra {
    /*
     * class body
     */
}
```

Terminating character

Semicolon (;) is the terminating character for any java statement.

Java Keywords



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

Variables and Datatypes



- A variable is a named storage location used to represent data that can be changed while the program is running
- A data type determines the values that a variable can contain and the operations that can be performed on it
- Categories of data types:
 - Primitive data types
 - Reference data types

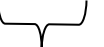
Reference Datatypes

- Reference data types represent objects
- A reference serves as a handle to the object, it is a way to get to the object
- Java has 2 reference data types
 - Class
 - Interface

Variable declaration and initialization

- Declaring a variable with primitive data type

```
int age = 21;
```

		
primitive	identifier	initial
type	name	value

- Declaring a variable with reference data type

```
Date now = new Date();  
String name = "Jason";
```

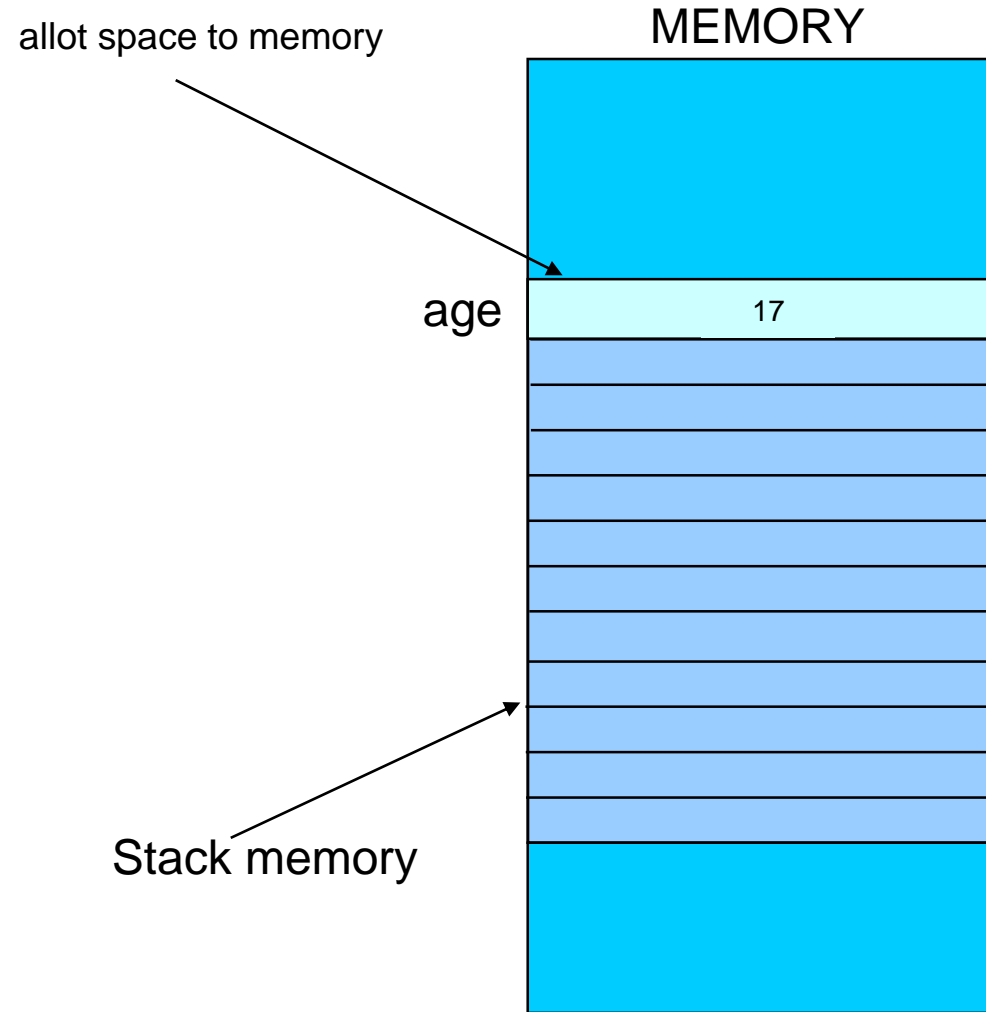
		
reference	identifier	initial
type	name	value

Primitive Type Declaration

`int age;`
type Identifier name

initialization/assignment

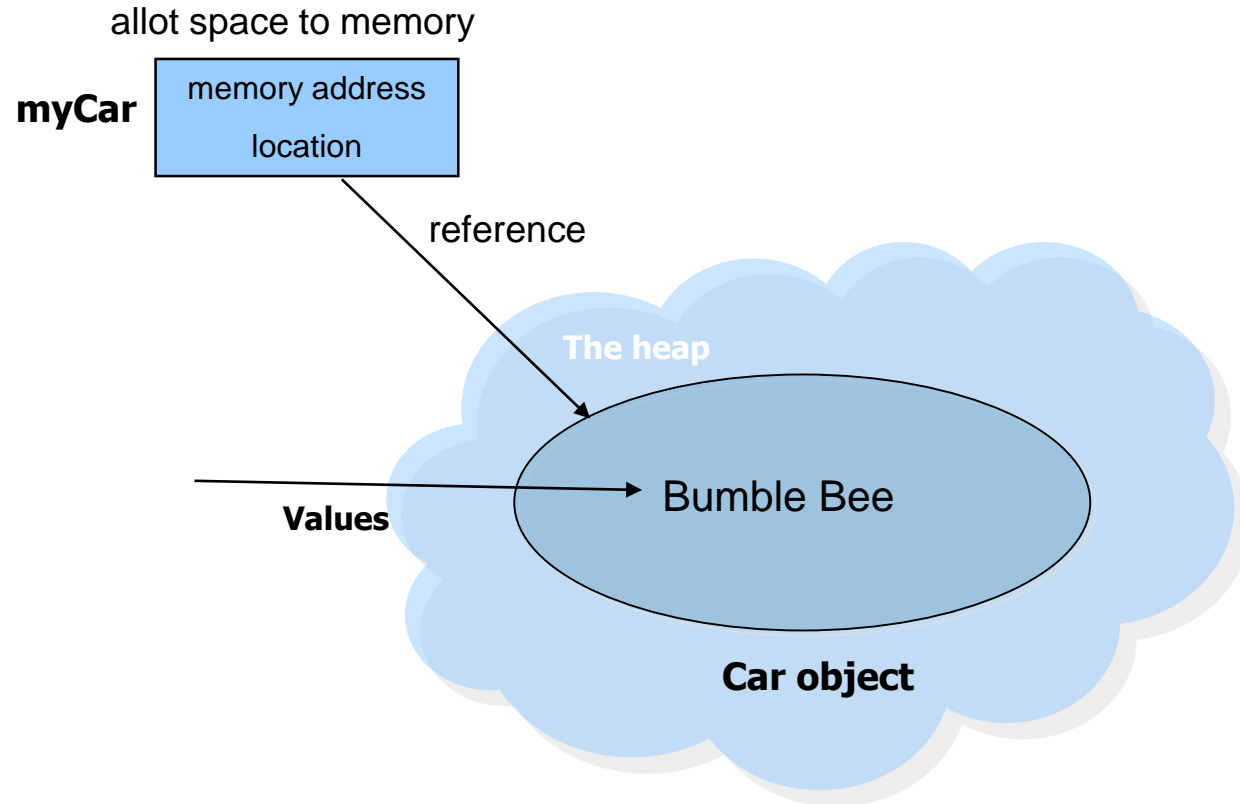
`age = 17;`
Identifier name value



Reference Type Declaration

`Car myCar;`
type Identifier name

Identifier name



Scope of variables

Member Variables

Declared inside the class but outside of all methods. Accessible by all methods of the class.

Local Variables Available only within the method where they were declared. Method parameters have local scope.

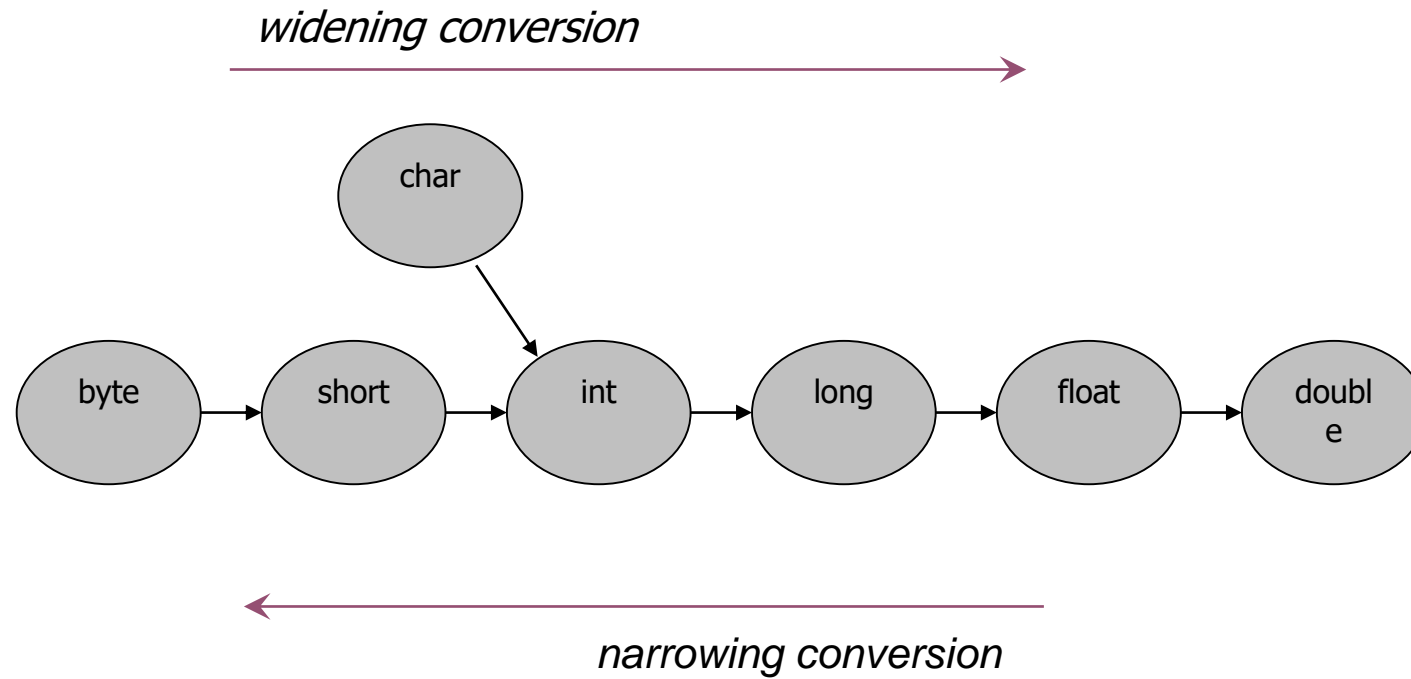
```
public class HelloWorld {  
    //accessible throughout the class  
    String name;  
  
    public void otherMethod(){  
        float salary = 15000.00f;  
        //can't access age variable from here  
    }  
  
    public static void main(String args[ ]) {  
        //can't access salary variable from here  
        int age=17;  
        //can't access ctr variable from here  
        for (int ctr=0 ; ctr<5 ; ctr++) {  
            //age variable accessible here  
        }  
    }  
}
```

Type Casting



- **Casting** is converting from one data type to another
 - Implicit casting is an implied casting operations
 - Explicit casting is a required casting operations
 - **Primitive casting** is converting a primitive data type to another
 - Widening conversion is casting a narrower data type to a broader data type
 - Narrowing conversion is casting a broader data type to a narrower data type
 - **Reference casting** is converting a reference data type to another
 - Upcasting is conversion up the inheritance hierarchy
 - Downcasting is conversion down the inheritance hierarchy
 - Casting between primitive and reference type is not allowed
 - In Java, casting is implemented using () operator

Type Casting



Flow Controls



- `if-else()` statement
- `switch()` statement
- `while()` statement
- `do-while()` statement
- `for()` statement
- `break` statement
- `continue` statement

Object Oriented Programming

Need for OOP Paradigm

- OOP is an approach to program organization and development, which attempts to eliminate some of the drawbacks of conventional programming methods by incorporating the best of structured programming features with several new concepts.
- OOP allows us to decompose a problem into number of entities called objects and then build data and methods (functions) around these entities.
- The data of an object can be accessed only by the methods associated with the object.

OOP Paradigm



- Emphasis is on data rather than procedure.
- Programs are divided into objects.
- Data Structures are designed such that they characterize the objects.
- Methods that operate on the data of an object are tied together in the data structure.
- Data is hidden and can not be accessed by external functions.
- Objects may communicate with each other through methods.

Class



- A class is a building block of OOP. It is the way to bind the data and its logically related functions together.
- An abstract data type that can be treated like any other built-in datatype.
- A class uses variables to define data fields and functions to define behaviors.
- Additionally, a class provides a special type of function, known as a **constructor**, which is invoked to create new objects from the class definition.

Methods

- A *method* refers to a piece of code referring to behaviors associated either with an object or its class
- A code found in a class for responding to a message
- The executable code that implements the logic of a particular message for a class
- An operation or function that is associated with an object and is allowed to manipulate the object's data

Methods

Steps in declaring a method

1. Set the return type
2. Provide method name
3. Declare formal parameters

method signature

- consists of the method name and its parameters
- must be unique for each method in a class

return statement

- allows the method to return a value to its caller
- also means to stop the execution of the current method and return to its caller
- implicit return at the end of the method

A method with empty parameters

A method that does not return a value must specify **void** as its return type

```
class Number {  
  
    int multiply(int i, int j) {  
        return i*j;  
    }  
  
    int divide(int i, int j) {  
        return i/j;  
    }  
  
    double getPi() {  
        return 3.14159265358979;  
    }  
  
    void printSum(int i, int j) {  
        System.out.println(i+j);  
    }  
}
```

Method calling

How to call a method

1. Method name should match
2. Number of parameters should match
3. Type of parameters should match

Ways of calling a method

1. Calling a method through its object name
2. Calling a method within the same class
3. Calling a static method through its class name

Method calling

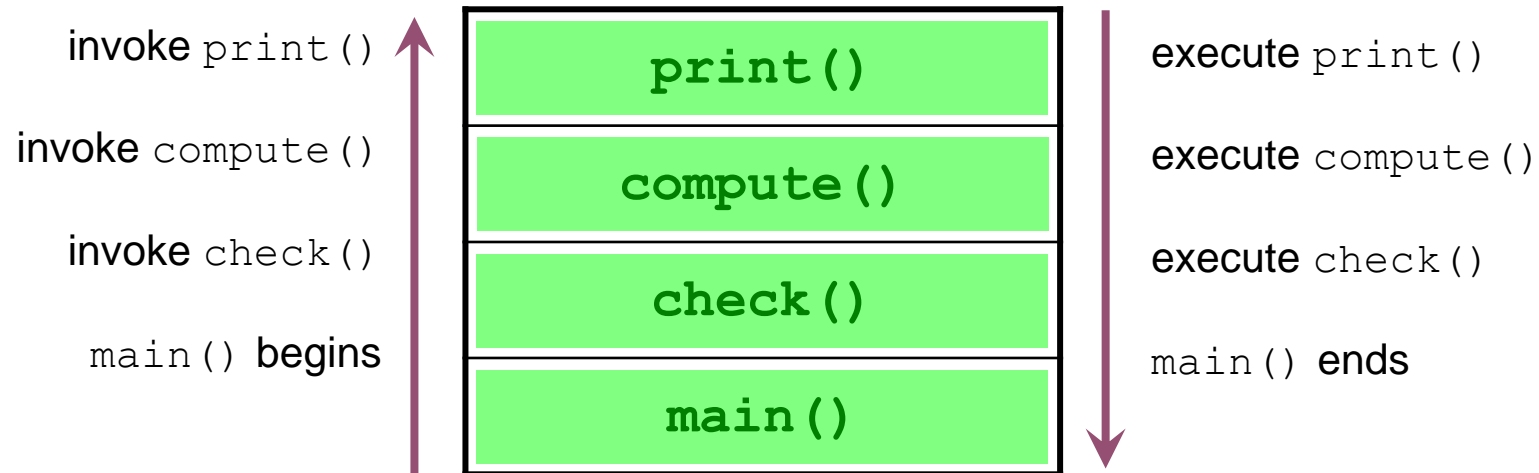
```
public class JavaMain {  
    public static void main(String[] args) {  
        // create a Person object  
        Person you = new Person();  
        you.talk();  
        you.jump(3);  
        System.out.println(you.tellAge());  
        JavaMain.talkOnly(you);  
        // create object of main program  
        JavaMain me = new JavaMain();  
        me.jumpOnly(you);  
    }  
    static void talkOnly(Person p) {  
        p.talk();  
    }  
    void jumpOnly(Person p) {  
        p.jump(2);  
    }  
}  
  
class Person {  
    void talk() {  
        System.out.println("blah, blah...");  
    }  
    void jump(int times) {  
        for (int i=0; i<times; i++) {  
            System.out.println("whoop!");  
        }  
    }  
    String tellAge() {  
        return "I'm " + getAge();  
    }  
    int getAge() {  
        return 10;  
    }  
}
```

The diagram illustrates the following method calls:

- `you.talk();` in `JavaMain` calls `talk()` in `Person`.
- `you.jump(3);` in `JavaMain` calls `jump(int times)` in `Person`.
- `JavaMain.talkOnly(you);` in `JavaMain` calls `talkOnly(Person p)` in `JavaMain`, which then calls `p.talk();` (referring to `you`), which calls `talk()` in `Person`.
- `me.jumpOnly(you);` in `JavaMain` calls `jumpOnly(Person p)` in `JavaMain`, which then calls `p.jump(2);` (referring to `you`), which calls `jump(int times)` in `Person`.
- `System.out.println(you.tellAge());` in `JavaMain` calls `tellAge()` in `Person`, which calls `getAge()` in `Person`.

Method call stack

- The Method Call Stack refers to all the methods currently active and being processed by a Java application



Passing parameters



Passing parameters in Java is always **Pass by Value!**

When passing a parameter of **primitive type**:

- A copy of the value of the variable is passed
- The passed variable cannot be changed in the called method as the method only possesses a copy of that variable.

When passing a parameter of **reference type**:

- A copy of the reference (address) of the object is passed
- The object reference cannot be changed in the called method (i.e., the object cannot be reassigned to another object)
- The object state can be changed in the called method (i.e., attributes can be modified)

Passing parameters

```
public class MySelf {
    public static void main(String[] args) {
        int age=18;
        int[] stats = {30,22,33};
        String name = "Mary Jane";
        MySelf me = new MySelf();

        System.out.println("I am just " + age);
        me.changeAge(age);
        System.out.println("Nope, I'd rather be " + age);
        System.out.println("I hate my body at " +
            me.getStats(stats));
        me.changeStats(stats);
        System.out.println("Wow! I'm now " +
            me.getStats(stats));
        System.out.println("My name is so naive, " +
            name);
        me.changeName(name);
        System.out.println("Naah, I still like " + name);
    }
}
```

```
void changeAge(int age) {
    age += 5;
    System.out.println("I imagine me being " + age);
}

void changeStats(int[] stats) {
    stats[0]=36; stats[1]=24; stats[2]=36;
    System.out.println("I wish I'm sexy at " + getStats(stats));
}

String getStats(int[] stats) {
    String s="";
    for (int i=0; i<stats.length; i++) s += stats[i] + " ";
    return s;
}

void changeName(String name) {
    String newName = "MJ";
    name = newName;
    System.out.println("What about " + name + "?");
}
}
```

Inheritance



Class hierarchy

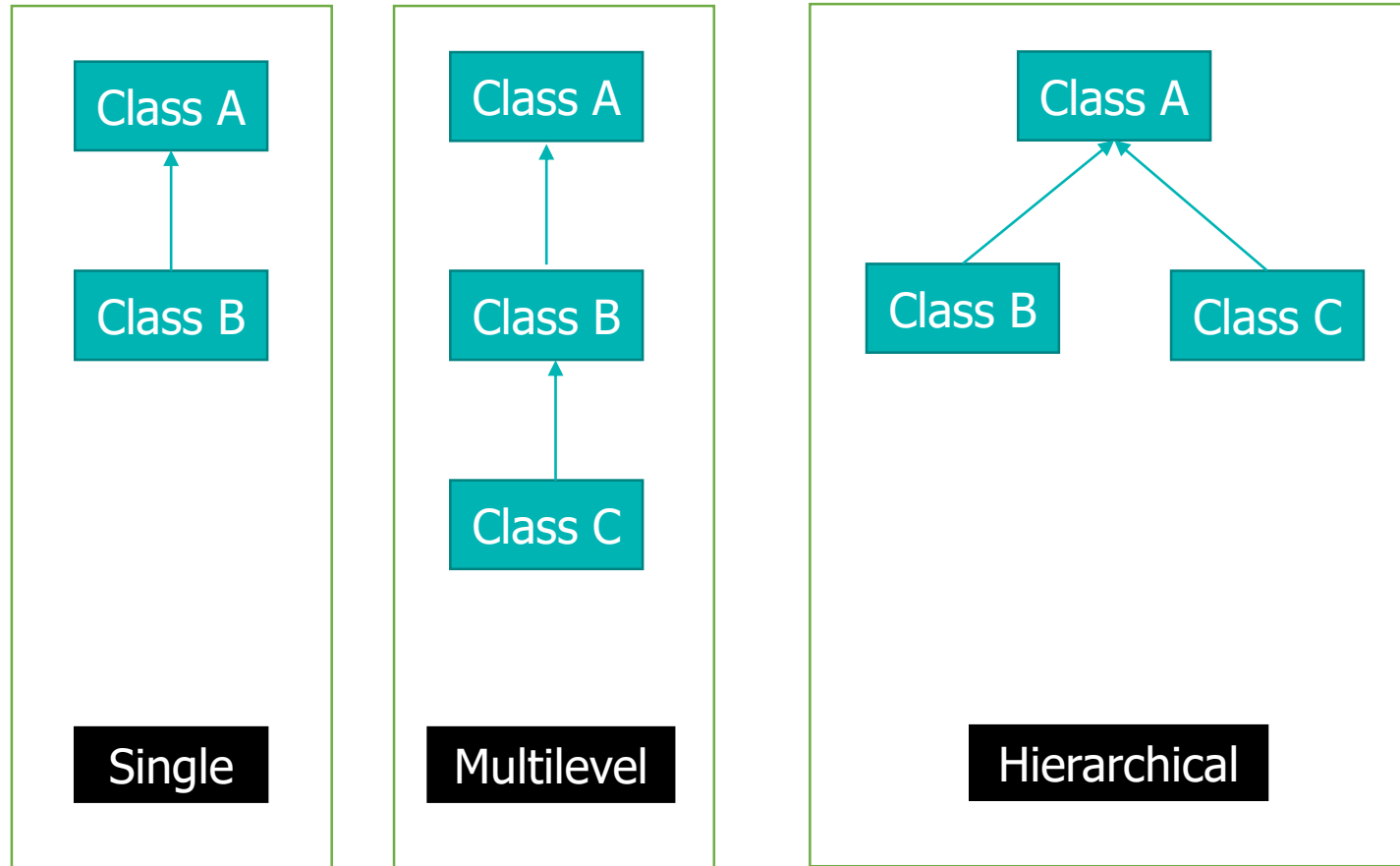
Inheritance is an IS-A relation

Generalization and Specialization

- subclass inherits attributes and services from its superclass
- subclass may add new attributes and services
- subclass may reuse the code in the superclass
- subclasses provide specialized behaviors (overriding and dynamic binding)
- partially define and implement common behaviors (abstract)

Multiple inheritance is not supported in Java

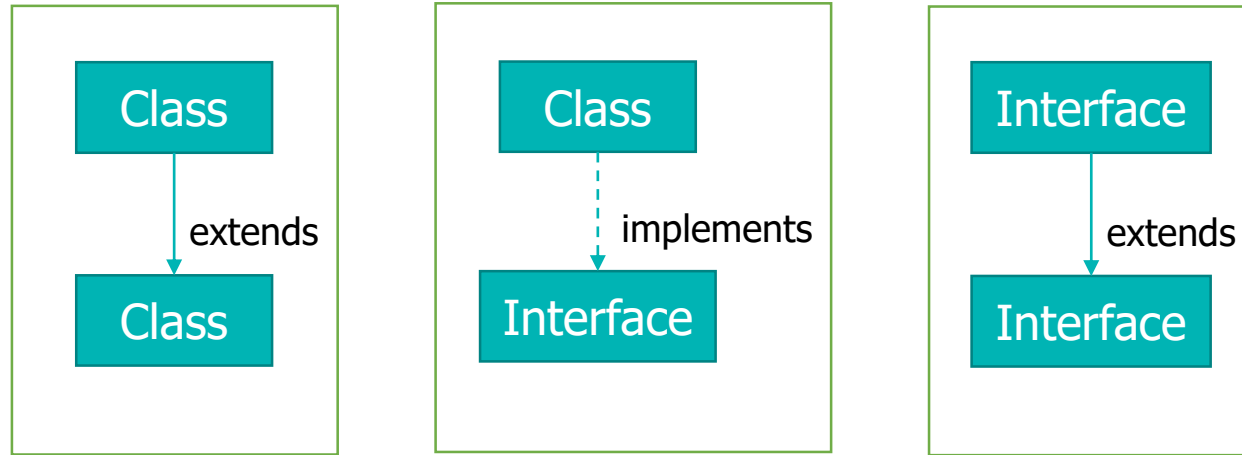
Types of Inheritance



Interfaces

- Interfaces are useful for the following:
 - Capturing similarities among unrelated classes without artificially forcing a class relationship
 - Declaring methods that one or more classes are expected to implement
 - Revealing an object's programming interface without revealing its class
- When to use an interface?
 - Perfect tool for encapsulating the classes inner structure.
 - Only the interface will be exposed

Interfaces and class



Arrays in Java

Array in Java

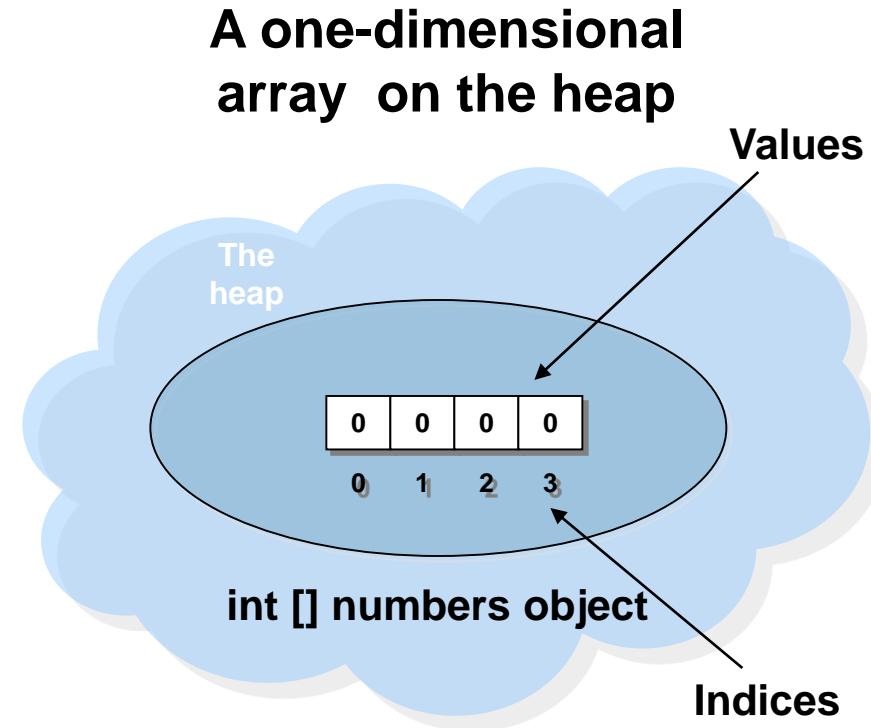
- An **array** is simply a sequence of either objects or primitives, **all of the same type** and packaged together under **one identifier name**.

`int[] numbers;`

type **Identifier name**

↓

Indexing operator



```
int numbers[] ;  
numbers = new int[4] ;
```

Creating an Array

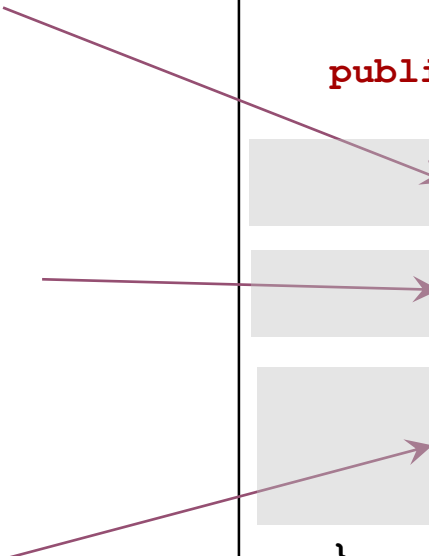
There are three steps to creating an array:

1. Declaration

2. Construction

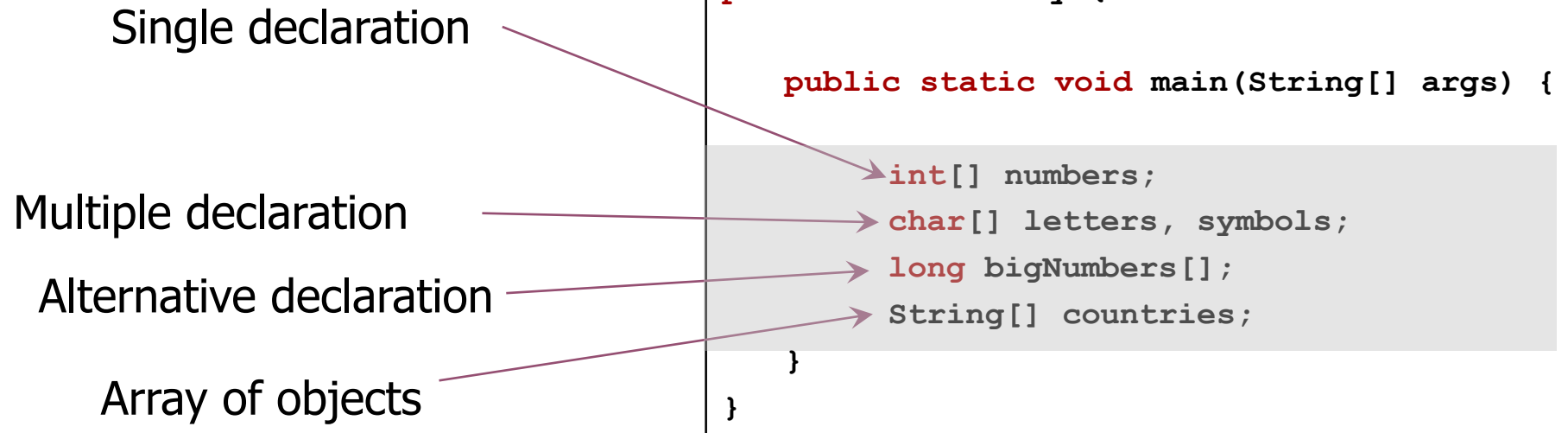
3. Initialization

```
public class Array {  
    public static void main(String[] args) {  
        int[] scores;  
        scores = new int[3];  
        scores[0] = 10;  
        scores[1] = 7;  
        scores[2] = 9;  
    }  
}
```



Array Declaration

Declaring an array means providing a name and its data type:



Constructing arrays

Constructing an array means creating an object of its declared type:

Create an array of `int`
type consisting of 3
elements

Declaration and
construction at the
same time

```
public class Array {  
  
    public static void main(String[] args) {  
  
        int[] numbers;  
        char[] letters, symbols;  
        long bigNumbers[];  
        String[] countries;  
  
        numbers = new int[3];  
        String[] currencies = new String[3];  
  
    }  
}
```

Array Initializations

Initializing an array means assigning values to its elements:

Array index starts with 0

Declaration,
construction,
initialization at
the same time

```
public class Array {  
  
    public static void main(String[] args) {  
  
        int[] numbers;  
        char[] letters, symbols;  
        long bigNumbers[];  
        String[] countries;  
  
        numbers = new int[3];  
        String[] currencies = new String[3];  
  
        numbers[0]=100;  
        numbers[1]=200;  
        numbers[2]=300;  
        int[] newNumbers = {1,2,3};  
    }  
}
```

Manipulating Array

length gives the size of the array

Printing each element of an array

Passing array to a method

Passing anonymous array

Assigning array to another array

Sample Output

```
100
200
300
600
6
```

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

        int[] numbers = new int[3];

        numbers[0]=100;
        numbers[1]=200;
        numbers[2]=300;
        int[] newNumbers = {1,2,3};

        for (int i=0; i<numbers.length; i++) {
            System.out.println(numbers[i]);
        }

        sumNumbers(numbers);
        sumNumbers(new int[]{3,2,1});
        numbers = newNumbers;
    }

    static void sumNumbers(int[] n) {
        int sum=0;
        for (int i=0; i<n.length; i++) {
            sum += n[i];
        }
        System.out.println(sum);
    }
}
```

Introduction to Java API



Introduction Java API



- **java.lang** package
 - Object class
 - Class class
 - System class
 - String and StringBuffer classes
 - Math class
 - Wrapper classes
- **java.util** package

java.lang package

- java.lang provides classes that are fundamental to the design of the Java programming language.
 - Object class, the root of the class hierarchy.
 - Class class, represents classes at run time.
 - Wrapper classes represent primitive types as objects.
 - Math class provides mathematical functions.
 - String and StringBuffer classes provide operations on strings.
 - System classes provide system operations.
 - Throwable class represents errors and exceptions.
- java.lang is implicitly imported in every Java source file.

Object class

Declaration: public class **Object**

- Class Object is the root of the class hierarchy. Every class has Object as a superclass.
- All objects inherit the methods of this class.

Method Summary		
protected	Object	clone()
	boolean	equals (Object obj)
protected	void	finalize()
	Class <? extends Object >	getClass()
	int	hashCode()
	void	notify()
	void	notifyAll()
	String	toString()
	void	wait()
	void	wait (long timeout)
	void	wait (long timeout, int nanos)

Class class



Declaration: public final class **Class** extends Object
implements Serializable, GenericDeclaration, Type, AnnotatedElement

- Instances of the class Class represent classes and interfaces in a running Java application.
- Every array also belongs to a class that is reflected as a Class object that is shared by all arrays with the same element type and number of dimensions.
- The primitive Java types (boolean, byte, char, short, int, long, float, and double), and the keyword void are also represented as Class objects.
- Class has no public constructor. Instead Class objects are constructed automatically by the Java Virtual Machine as classes are loaded in the class loader.

System class

Declaration: `public final class System extends Object`

- The `System` class contains several useful class fields and methods which are related to the following operations:
 - standard input, standard output, and error output streams.
 - access to externally defined properties and environment variables.
 - a means of loading files and libraries.
 - and a utility method for quickly copying a portion of an array.

System class

Field Summary

static PrintStream	err
static InputStream	in
static PrintStream	out

Method Summary (Partial List)

static void	arraycopy (Object src, int srcPos, Object dest, int destPos, int length)
static String	clearProperty (String key)
static long	currentTimeMillis ()
static void	exit (int status)
static void	gc ()
static Map < String , String >	getenv ()
static String	getenv (String name)
static Properties	getProperties () .
static SecurityManager	getSecurityManager ()
static void	load (String filename)
static long	nanoTime ()
static void	runFinalization ()
static void	runFinalizersOnExit (boolean value)
static void	setErr (PrintStream err)
static void	setIn (InputStream in)
static void	setOut (PrintStream out)
static void	setProperties (Properties props)
static void	setSecurityManager (SecurityManager s)

String class

Declaration:

public final class **String** extends Object implements Serializable, Comparable<String>, CharSequence

- The String class represents character strings. All string literals in Java programs are implemented as instances of this class.
- Strings are immutable, their values cannot be changed after they are created
- The class String includes methods for examining individual characters of the sequence, for comparing strings, for searching strings, for extracting substrings, and for creating a copy of a string with all characters translated to uppercase or to lowercase.

Method Summary (Partial List)	
char	charAt (int index)
int	compareTo (String anotherString)
String	concat (String str)
boolean	contains (CharSequence s)
boolean	endsWith (String suffix)
boolean	equals (Object anObject)
static String	format (String format, Object... args)
int	hashCode ()
int	indexOf (int ch)
int	length ()
boolean	matches (String regex)
String	replace (char oldChar, char newChar)
String []	split (String regex)
boolean	startsWith (String prefix)
String	substring (int beginIndex)
String	toLowerCase ()
String	toUpperCase ()
String	trim ()

StringBuffer vs StringBuilder class

Declaration: public final class **StringBuffer**
extends Object implements
Serializable, CharSequence

- StringBuffer is a thread-safe, mutable sequence of characters.
- A StringBuffer is like a String, but can be modified.
- StringBuffer class has been supplemented with an equivalent class designed for use by a single thread, StringBuilder.
- The StringBuilder class should generally be used in preference to StringBuffer as it supports all of the same operations but is faster as it performs no synchronization.

Method Summary (Partial List)

StringBuffer	append (String str)
StringBuffer	append (StringBuffer sb)
int	capacity ()
char	charAt (int index)
StringBuffer	delete (int start, int end)
StringBuffer	deleteCharAt (int index)
int	indexOf (String str)
StringBuffer	insert (int offset, String str)
int	lastIndexOf (String str)
int	length ()
StringBuffer	replace (int start, int end, String str)
StringBuffer	reverse ()
void	setCharAt (int index, char ch)
void	setLength (int newLength)
String	substring (int start)
String	substring (int start, int end)
String	toString ()
void	trimToSize ()

Math class

Declaration: public final class **Math** extends Object

- Math class contains methods for performing basic numeric operations such as elementary exponential, logarithm, square root, and trigonometric functions.
- Math class cannot be extended (it is declared final) nor instantiated (its constructor is private).
- Its methods are declared static and can be invoked using its class name.

Method Summary (Partial List)

static double	abs (double a)
static double	ceil (double a)
static double	cos (double a)
static double	exp (double a)
static double	floor (double a)
static double	log (double a)
static double	log10 (double a)
static double	log1p (double x)
static double	max (double a, double b)
static double	min (double a, double b)
static double	pow (double a, double b)
static double	random ()
static long	round (double a)
static double	sin (double a)
static double	sqrt (double a)
static double	tan (double a)
static double	toDegrees (double angdeg)
static double	toRadians (double angdeg)

Wrapper classes

The wrapper classes serve two primary purposes:

To provide a mechanism to "wrap" primitive values in an object so that the primitives can be included in activities reserved for objects, such as being added to collections, or returned from a method with an object return value.

To provide an assortment of utility functions for primitives. Most of these functions are related to various conversions: converting primitives to and from String objects, and converting primitives and String objects to and from different bases (or radix), such as binary, octal, and hexadecimal.

Wrapper class constructor arguments



Primitive	Wrapper Class	Constructor Arguments
boolean	Boolean	boolean or String or null
byte	Byte	byte of String
char	Character	char
double	Double	double or String
float	Float	float, double, or String
int	Integer	int or String
long	Long	long or String
short	Short	short or String

java.util package

Contains classes related to the following:

- Collections framework
- Legacy collection classes
- Event model
- Date and time facilities
- Internationalization
- Miscellaneous utility classes

Exception Handling



What is an Exception?

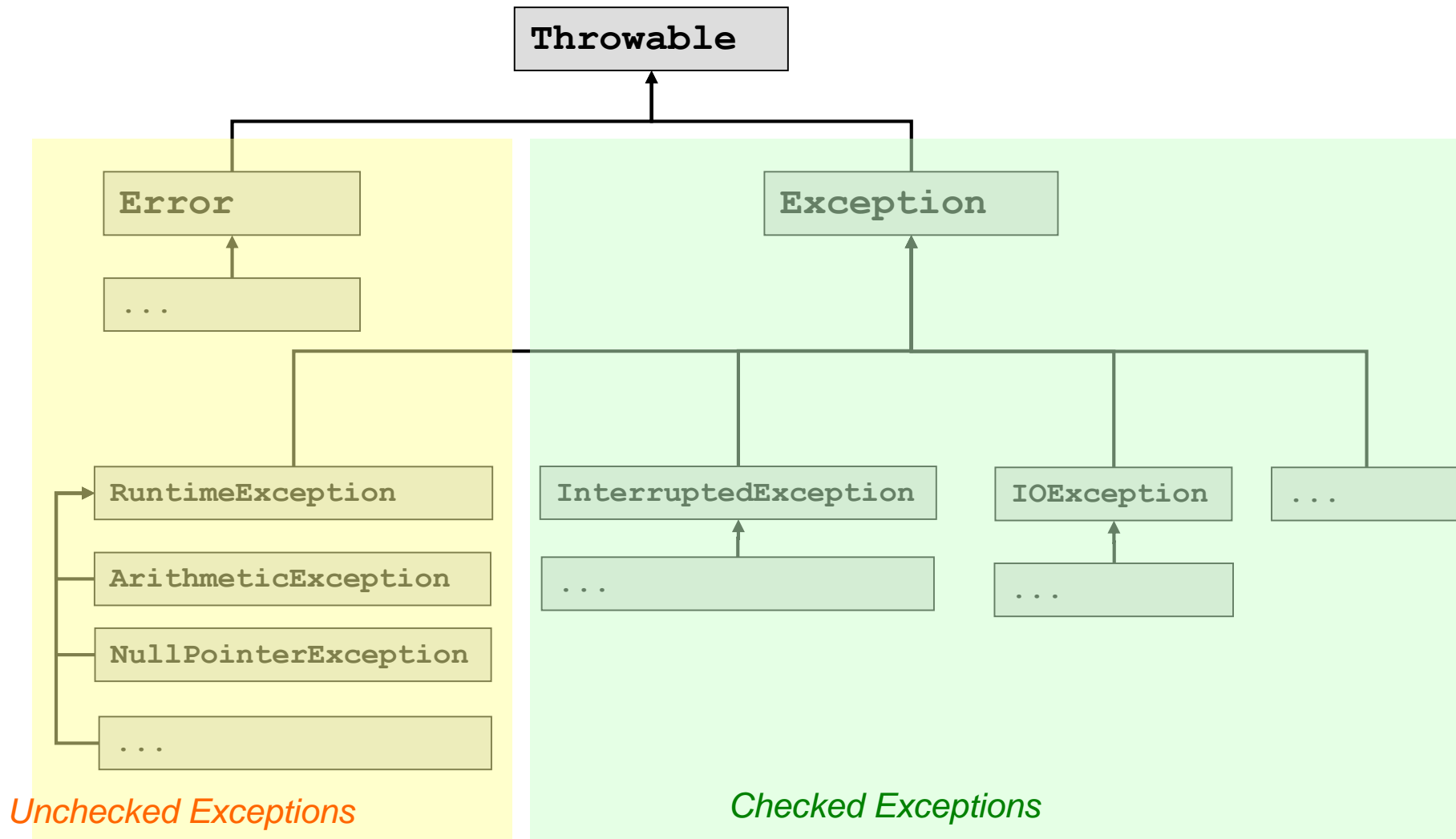
- An event during program execution that prevents the program from continuing normally
- An error condition that changes the normal flow of control in a program
- A signal that some unexpected condition has occurred in the program

Exception Handling Mechanism



- Exception mechanism is built around the throw-and-catch paradigm
 - “to throw” means an exception has occurred
 - “to catch” means to deal with an exception
- If an exception is not caught, it is propagated to the call stack until a handler is found
- Propagating an exception is called “ducking” the exception, or “passing the buck”

Exception class Hierarchy



Types of Exceptions

- All exceptions in Java are objects of Throwable class
- Unchecked Exceptions
 - are exceptions derived from Error and RuntimeException classes
 - are usually irrecoverable and not handled explicitly
 - are not checked by the compiler
- Checked Exceptions
 - are exceptions derived from Exception class excluding the RuntimeException class
 - must be handled explicitly
 - are checked by the compiler
- Both checked and unchecked exceptions can be thrown and caught
- New exceptions are created by extending the Exception class or its subclasses

try-catch-finally

```
try {  
    /*  
     * some codes to test here  
     */  
} catch (Exception1 ex) {  
    /*  
     * handle Exception1 here  
     */  
} catch (Exception2 ex) {  
    /*  
     * handle Exception2 here  
     */  
} catch (Exception3 ex) {  
    /*  
     * handle Exception3 here  
     */  
} finally {  
    /*  
     * always execute codes here  
     */  
}
```

`try` block encloses the context where a possible exception can be thrown

each `catch()` block is an exception handler and can appear several times

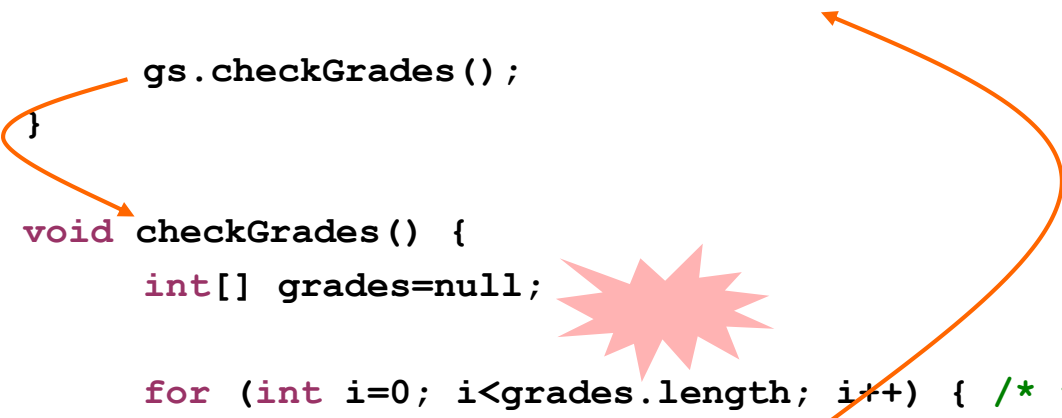
Exception1 should not shadow *Exception2* which in turn should not shadow *Exception3* (based on the exception hierarchy)

`finally` block is always executed before exiting the `try` statement. `finally` block is optional but must appear once after the `catch()` blocks

at least one `catch()` block or `finally` block must appear in the `try` statement

Unhandled Exceptions

```
public class GradingSystem {  
  
    public static void main(String[] args) {  
        GradingSystem gs = new GradingSystem();  
  
        gs.checkGrades();  
    }  
  
    void checkGrades() {  
        int[] grades=null;  
  
        for (int i=0; i<grades.length; i++) { /* test here*/ };  
    }  
}
```



The diagram illustrates the execution flow and the point of failure. An orange arrow originates from the `gs.checkGrades();` line in the `main` method and points to the `checkGrades()` method definition. Another orange arrow starts from the `for` loop in `checkGrades()` and points back to the `new GradingSystem();` line in `main`. A pink starburst icon is placed over the `grades` variable in the `for` loop, indicating the location of the `NullPointerException`.

```
Exception in thread "main" java.lang.NullPointerException  
    at  
    codesnippets.GradingSystem.checkGrades(GradingSystem.java:31  
    )  
    at codesnippets.GradingSystem.main(GradingSystem.java:6)
```

Handling Exceptions



```
public class GradingSystem {  
  
    public static void main(String[] args) {  
        GradingSystem gs = new GradingSystem();  
  
        gs.checkGrades();  
    }  
  
    void checkGrades() {  
        int[] grades=null;  
  
        try {  
            for (int i=0; i<grades.length; i++) { /* test here*/ };  
        } catch (NullPointerException e) {  
            System.out.println("Grades may be empty!");  
        } catch (RuntimeException e) {  
            System.out.println("Problem while executing!");  
        } catch (Exception e) {  
            System.out.println("Error in checking grades!");  
        } finally {  
            System.out.println("Finished checking grades.");  
        }  
    }  
}
```

Grades may be empty!
Finished checking grades.

throws and throws

```
void validate() throws Exception3 {  
    try {  
        // throw an exception  
        throw new Exception3();  
  
    } catch (Exception1 ex) {  
        /*  
         * handle Exception1 here  
         */  
    } catch (Exception2 ex) {  
        /*  
         * handle Exception2 here  
         */  
    } catch (Exception3 ex) {  
        /*  
         * can't handle Exception3,  
         * propagate to call stack  
         */  
        throw new Exception3(ex);  
    }  
}
```

`throws` clause declares the unhandled exceptions that a method can throw. It propagates the exception to the calling method. It becomes part of the method signature

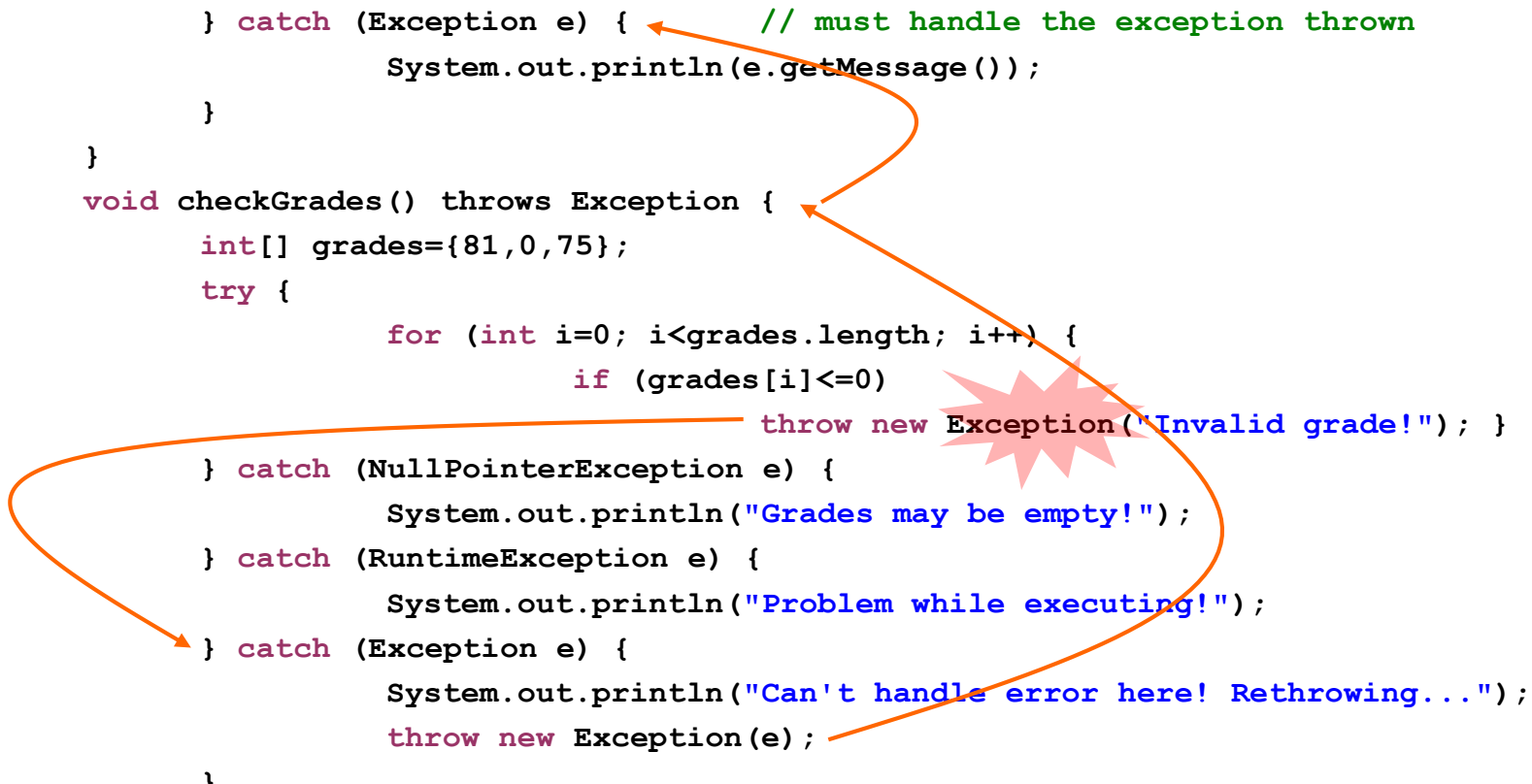
Any code that calls this method must handle the exception declared in the `throws` clause

`throw` statement explicitly throws an exception

`throw` statement propagates (rethrows) a caught exception

Propagating Exceptions

```
public class GradingSystem {
    public static void main(String[] args) {
        GradingSystem gs = new GradingSystem();
        try {
            gs.checkGrades();
        } catch (Exception e) { // must handle the exception thrown
            System.out.println(e.getMessage());
        }
    }
    void checkGrades() throws Exception {
        int[] grades={81,0,75};
        try {
            for (int i=0; i<grades.length; i++) {
                if (grades[i]<=0)
                    throw new Exception("Invalid grade!"); }
        } catch (NullPointerException e) {
            System.out.println("Grades may be empty!");
        } catch (RuntimeException e) {
            System.out.println("Problem while executing!");
        } catch (Exception e) {
            System.out.println("Can't handle error here! Rethrowing...");
            throw new Exception(e);
        }
    }
}
```



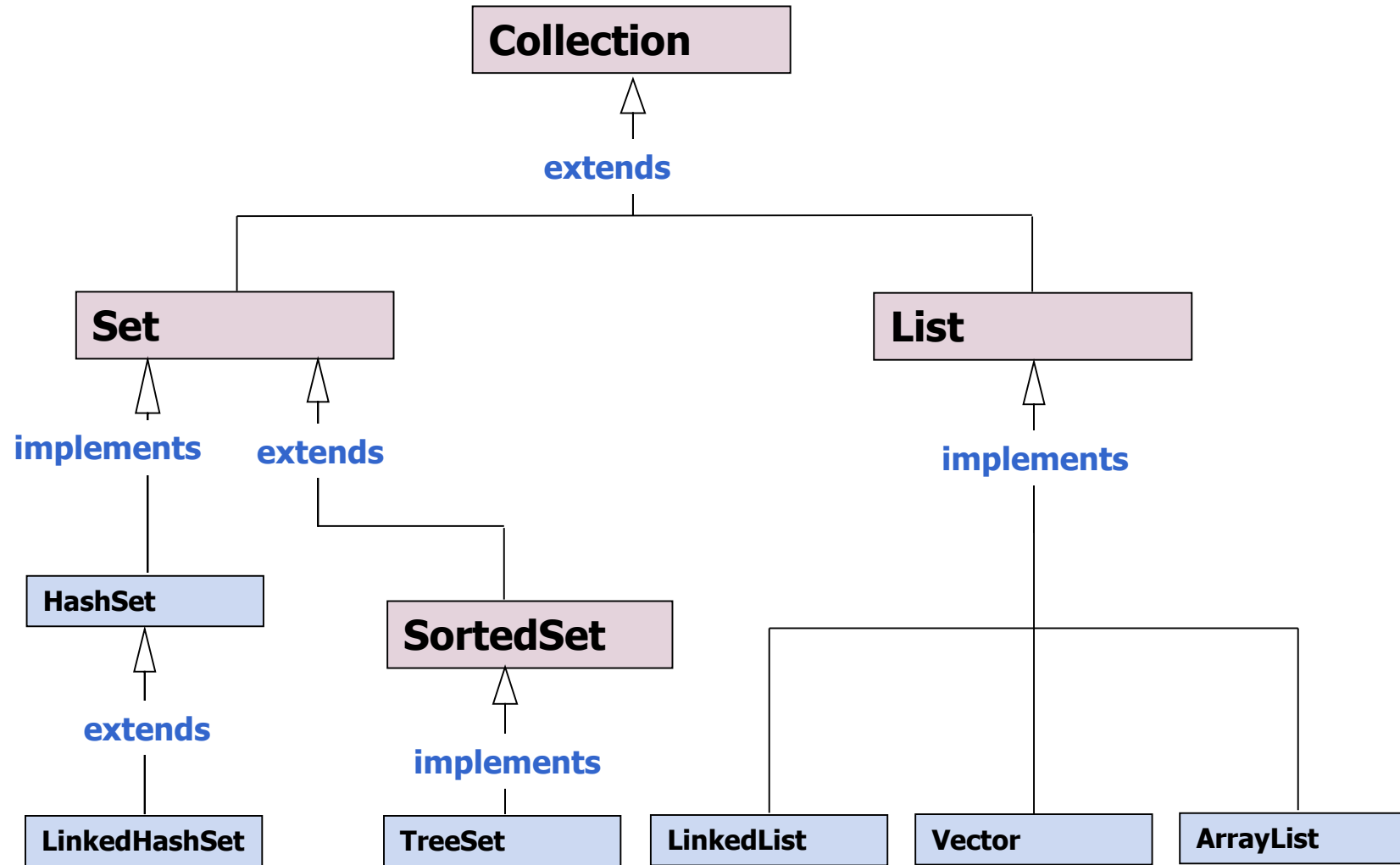
Collection Framework



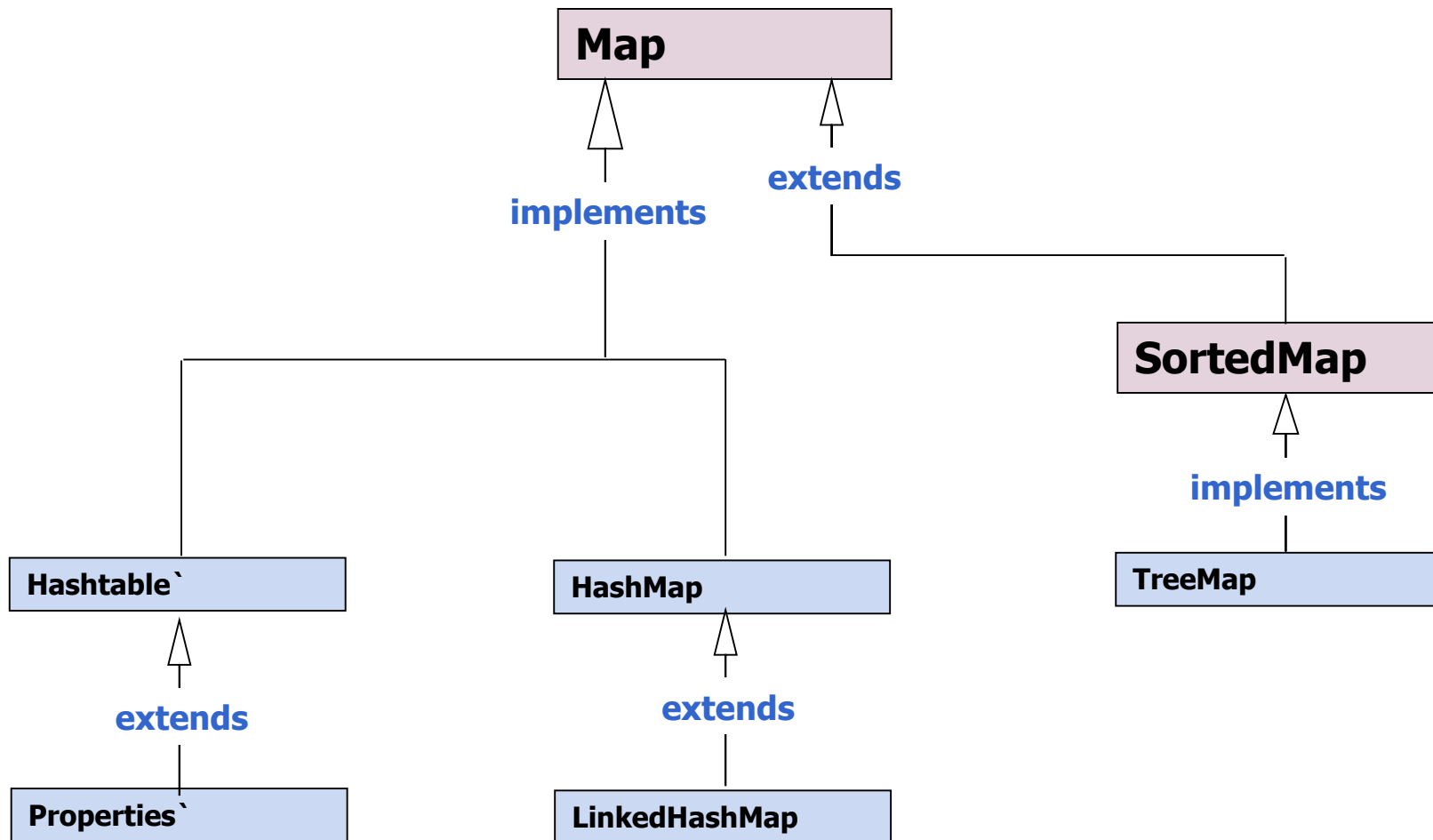
What is a Collection

- A **Collection** (also known as container) is an object that contains a group of objects treated as a single unit.
- Any type of objects can be stored, retrieved and manipulated as elements of collections.

Collection class Hierarchy



Map class Hierarchy



Collection Interfaces



Core Interface	Description
Collection	specifies contract that all collections should implement
Set	defines functionality for a set of unique elements
SortedSet	defines functionality for a set where elements are sorted
List	defines functionality for an ordered list of non- unique elements
Map	defines functionality for mapping of unique keys to values
SortedMap	defines functionality for a map where its keys are sorted

Collection Implementations



Set	List	Map
HashSet	ArrayList	HashMap
TreeSet	LinkedList	TreeMap
	Vector	Hashtable
		Properties

Collection Operations



Basic Collection Operations

- Check if collection is empty
 - Check if an object exists in collection.
 - Retrieve an object from collection
 - Add object to collection
 - Remove object from collection
 - Iterate collection and inspect each object
-
- Each operation has a corresponding method implementation for each collection type

Collection Characteristics

- Ordered
 - Elements are stored and accessed in a specific order
- Sorted
 - Elements are stored and accessed in a sorted order
- Indexed
 - Elements can be accessed using an index
- Unique
 - Collection does not allow duplicates

Iterator



- An *iterator* is an object used to mark a position in a collection of data and to move from item to item within the collection

Syntax:

```
Iterator <variable> = <CollectionObject>.iterator();
```

Lambda Expression & Streams



Lambda Expression



- Lambda expression enables functional programming in Java
- Enable you to write more readable, maintainable and concise code
- **Why Lambdas ?**
 - In java it was not possible to pass a block of code to some method. If you want to do so, create an object belonging to a class that contains the code block and pass the object.
 - Instead of this approach, we can use concise lambda expressions in Java -8
- You can supply a lambda expression whenever an object of an interface with a Single Abstract Method is expected. Such interfaces are called Functional interfaces.

- **Example**

```
Comparator<String> lenComp = (name1, name2) -> { return Integer.compare(name1.length(), name2.length()); }  
Collections.sort(nameList, lenComp);
```

Streams – Java 8

- Streams brings functional programming to Java and are supported starting in Java 8.
- Advantages of Streams
 - Will make you a more efficient programmer
 - Make heavy use of Java Lambda expressions
 - Parallel streams makes it easy to multi-threaded operations

Streams

- A stream pipeline consists of a source, followed by zero or more intermediate operations; and a terminal operation.



- Stream source
 - Streams can be created from Collection, List, Set, Arrays, lines of file
- Stream operations are either intermediate or terminal.
 - Intermediate operations returns a stream, so that we can chain multiple intermediate operations
 - Terminal operations returns either void or non-stream result.

Streams

Intermediate operations

- Zero or more intermediate operations allowed.
- Order matters for large datasets. Filter first and then map or sort etc.
- For very large datasets use parallel streams to enable multiple threads.
- Some of the intermediate operations

`anyMatch()`

`distinct()`

`filter()`

`findFirst()`

`flatMap()`

`map()`

`skip()`

`sorted()`

Streams



Terminal operations

- One terminal operations allowed.
 - `forEach()` applies the same function to each element
 - `collect()` saves the elements into a collection
 - Other options reduce the stream to a single summary element

<code>count()</code>	<code>min()</code>
<code>max()</code>	<code>reduce()</code>
	<code>summaryStatistics()</code>

Java 8 Date/Time API



Java 8 Date/Time API



- Java 8 introduced new APIs for *Date* and *Time* to address the shortcomings of the older *java.util.Date* and *java.util.Calendar* APIs
- **LocalDate/LocalTime** and **LocalDateTime** API : Use it when time zones are NOT required.
- **ZonedDateTime** : Use it when we need timezone specific date and time
- **Period** and **Duration** classes
- Refer <https://www.baeldung.com/java-8-date-time-intro> for more information

IO Streams & Loggers



Files and directory management in Java

- File object represents a file or directory in the file system.
- Defined in the java.io package
- Creating file objects

```
File file = new File("filename");  
File folder = new File(String path);  
File file = new File(File path, String filename);
```
- Example:

```
File file = new File("test.txt");  
File myfolder = new File("d:/myfolder");  
File file2 = new File(myfolder, "example.txt");
```

Input-Output Streams

- Stream is a flow of data (byte/char) from a source to a destination.
- Source can be from a keyboard, file, network etc.
- Destination can be console, file, network etc.
- Separate streams are available for both input and output.
- Java implements streams within a class hierarchy defined in **java.io** package.
- All input-output operations throws IOException, which is a checked exception, must be handled.

Pre-defined streams

- `System.out` – Represents the standard output device(Screen)
- `System.in` – Represents the standard input device(Keyboard)
- `System.err` - Represents the standard error device(Screen)

Reading from Files



Reading text files

```
FileReader fr = new FileReader(String filepath);
```

```
FileReader fr = new FileReader(File file);
```

Reading in binary format

```
FileInputStream fin = new FileInputStream(String filepath);
```

```
FileInputStream fin = new FileInputStream(File file);
```

It throws FileNotFoundException.

Reading from Files

BufferedReader improves read performance by buffering input.

```
BufferedReader br = new BufferedReader(new FileReader("filename"));  
String content = br.readLine();
```

Writing to a File

FileWriter class creates Writer that you can use to write to a file.

```
FileWriter fw = new FileWriter(String filepath);
```

```
FileWriter fw = new FileWriter(String filepath, boolean append);
```

```
FileWriter fw = new FileWriter(File file, boolean append);
```

```
fw.write(char);
```

```
fw.write(String)
```

Writing to a file

PrintWriter class provides formatted output to a file.

```
Printwriter pw = new PrintWriter(new FileWriter(String filepath));  
pw.println("content");  
pw.printf("%s %.2f", name, amount);
```

Reading from System.in



Method-1

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));  
String input = br.readLine();
```

Method-2

```
Scanner scan = new Scanner(System.in);  
String input = scan.nextLine();
```


Loggers – Log4j

- Logging is the process of writing log messages during program execution to a central place.
- These messages can later be retrieved and analyzed.
- A well-written logging code offers quick debugging, easy maintenance, and structured storage of application's runtime information.

Log4j - components

- Log4j has three main components.
- **Loggers** -
For capturing log information.
- **Appenders** -
Publish logs to various destinations (Usually file, stdout)
- **Layout** - Formats log messages.

Setting-up Log4j

- Add log4j.jar to the project build path
- log4j.properties file is a configuration file which should be added to the classpath.
- Configuration file contains level, appenders and layout information.

Logging - Level

- Log levels defines the severity of the log message.

- Log4j has following Levels in descending order
 - FATAL
 - ERROR
 - WARN
 - INFO
 - DEBUG
 - TRACE

Logging messages



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

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

```
logger.info("information message");
```

```
logger.error("error message");
```

Java Database Connectivity



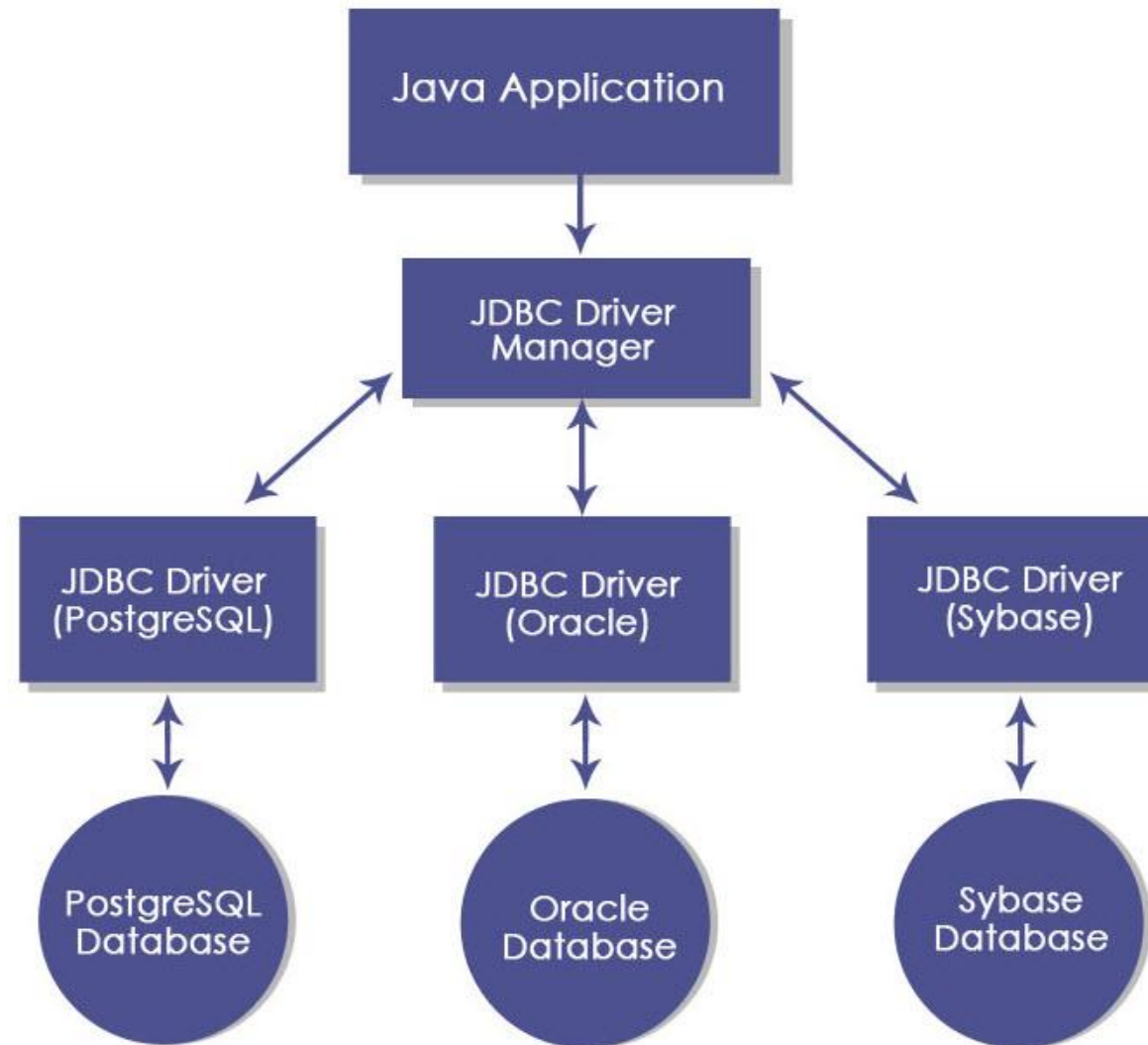
JDBC - API



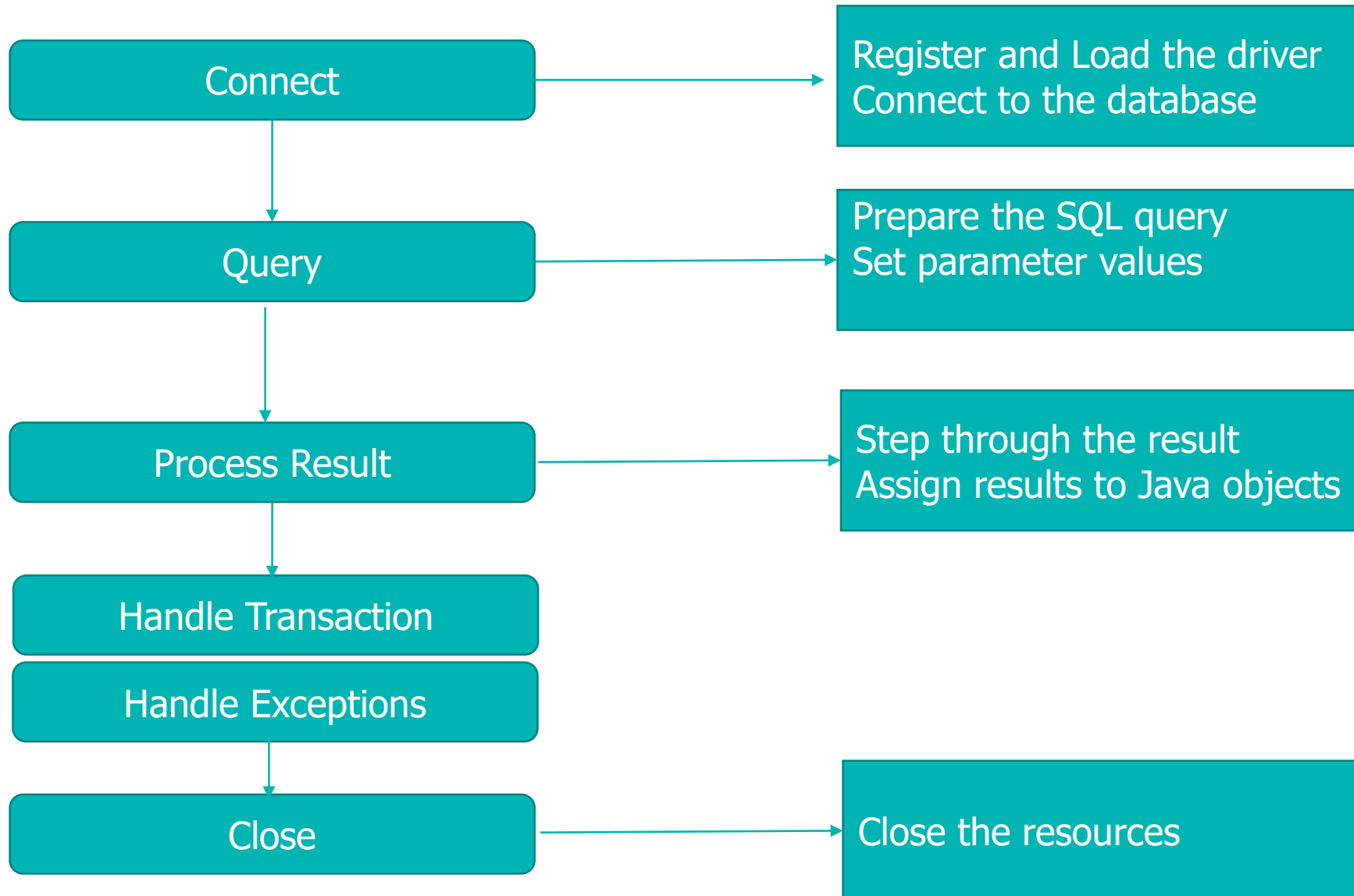
What is JDBC

- JDBC API allows java program to communicate with RDBMS.
- We can manipulate data in RDBMS tables from within Java programs.
- JDBC API supports both two tier and three tier models for data access
- In Two tier Model – Java Application directly interacts with database
- In Three tier Model – Introduces a middle-level server, it maintains control over data access
- JDBC API is defined in two packages
 java.sql and javax.sql package

JDBC Architecture`



Steps to write JDBC programs



Data Access Object Pattern

- Data Access Object Pattern or DAO pattern is used to separate low level data accessing API or operations from high level business services
- Parts of a DAO
 - Data Access Object Interface
 - Data Access Object Interface Implementation class
 - Model object or Value object

