# Java for .NET Programmers
# -
# Objects and the Platform API

**Student Workbook 2**

Version 2023.02.07

**Presented to**
**The Harford Insurance Group**
**8 February, 2023**


**Author**

Paul Kimball
Interface Associates

# Table of Contents

# Module 1

# Classes and Objects

# `static` Members

- **`static` members are associated with the class itself**
  - `static` methods are called "*class methods*"
  - `static` variables are called "*class variables*"

- **`static` members are allocated in memory and become accessible as soon as a class is loaded into the JVM**

- **For clarity, always use the class name in front of the dot when accessing `static` members**

**Example**

```
package com.pekia.examples;

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

    double d = Math.sqrt(2);
    double a = Math.PI * Math.pow(4,2);  // pi * 4^2

    System.out.println("Diagonal of 1x1 square: " + d);
    System.out.println("Area of circle, rad 4: " + a);
  }
}
```

  - Class exercise: look up `java.lang.Math` in the online docs. What `static` methods and constants does it have?

# Static Import (Java SE 5+)

- **Many classes have useful `static` members**

- **The `import static` statement can be used to import `static` members of a class**
  - Allows access with even shorter names
  - Wildcard * imports all `static` variables and methods
    - Beware of naming conflicts with local variables and methods

**Example**

```
package com.pekia.examples;

import static java.lang.System.out; // one static member
import static java.lang.Math.*;     // all static members

public class GeometryCalculator2 {

  public static void main(String[] args) {

    double d = sqrt(2);           // Math static methods
    double a = PI * pow(4,2);     // pi * 4^2

    out.println("Diagonal of 1x1 square: " + d);
    out.println("Area of circle, radius 4: " + a);
  }
}
```

# Objects

- **An object is an *instance* of a class**
  - Has a unique identity and state
  - Has its own copy of all non-`static` member variables (*instance variables*)

- **An object is a *reference type***
  - The `new` operator returns a reference, which can be assigned to a variable

**Example**

```
// s holds a reference to a String object

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

- **`null` is a legal value for a reference**
  - Useful for testing to see if an instance has been created or not
    - If used to access methods or variables, it causes a `NullPointerException`

**Example**

```
String message = null;  // initialize reference
...
if ( message == null ) {
  // Create a String and assign the reference
  message = new String("Hello World");
}
System.out.println(message);
```

# Using Objects

- **Declare a reference variable**

```
import java.util.*;
...
Date d;                // Declare a reference
d = null;              // null is an OK value
```

- **Create an object with the `new` operator**

```
if (d == null)  d = new Date();
```

- **Access methods and fields using the dot**

```
long ms = d.getTime();     // msecs since Jan 1, 1970
```

- **Pass the reference as an argument**

```
System.out.println( d );   // print the date
```

- **Return a reference from a method**

```
GregorianCalendar c = new GregorianCalendar();
Date d1 = c.getTime();
```

  - Note that `Date` and `GregorianCalendar` both have methods named `getTime()`. What is different about them?

# Comparing references

- **Compare references**

```
if (d != d1) System.out.println("different dates");
```

  - Operator checks if references are to the identical object. To compare the *values* of objects, use the `equals()` method

- **If the value of a reference variable is assigned to another variable, both references refer to the same object**

**Example**

```
String s1 = new String("There's only one World");
String s2 = s1;
```

- **When an object is created with `new`, the JVM calls its *constructor***
  - There may be several overloaded constructors distinguished by different parameter lists

```
Date d1 = new Date(630720000L);   // init with long
Date d2 = new Date();             // init with default value
```

- **The compiler determines which constructor to use based on the number and type of arguments**

# Memory in the JVM

- **The Stack**
  - Holds local variables and temporary variables, including method parameters and return values
  - A *Frame* is pushed onto the stack when a function is called, and popped when it returns
  - The JVM does not define the internal organization of stack memory
  - There is a separate stack for each thread

- **The Heap**
  - Holds memory that is dynamically allocated with `new`
  - Memory in the heap is freed by the *garbage collector*
  - Heap memory is shared between threads

- **Objects are *always* in the heap**

# Garbage Collection

- **The garbage collector frees an object from the heap when there are no remaining references to it**
  - Reference goes out of scope

```
{
  String str;
  str = new String("hello");
}
// str out of scope; String can be freed
```

  - Reference points to a different object

```
String str;
str = new String("hello");
str = new String("world");  // old String can be freed
```

  - Reference is set to `null`

```
String str = new String("hello");
str = null;                 // String can be freed
```

- **Objects cannot be freed explicitly**
  - However, you can call `System.gc()` to give the garbage collector a nudge

# Method Overloading

- **Within a class, multiple methods can have the same name as long as their parameter lists are different**
  - Methods with the same name are called *overloaded* methods

- **The appropriate method is called based on the method *signature***
  - Name of method must match
  - Number of parameters must match
  - Type order of parameters must match
  - Not part of the signature:
    - Names of parameters - they're just local variables in the method
    - Return type - value might not even be used by the caller

- **Compiler looks for an exact match first**
  - If not found, tries promoting args looking for a match

- **Method overloading is a form of "polymorphism" that must be resolved at compile time**
  - If compiler does not find a suitable match, code won't compile

# Some Useful Classes

- **java.lang.String**
- **java.lang.Math**
- **java.math.BigDecimal**
- **java.util.Date**
- **java.util.Calendar**
- **java.time.DateTime**

# `String`

- **`String` represents an *immutable* Unicode character string**
  - Value must be set at creation

```
String s = new String("T'was Brillig");
```

- **String literals  are used frequently, so they get some special treatment by the JVM**
  - These are created in a part of the heap called the *string pool*; there is only a single copy per literal value

```
  s1 = "Slithey Toves";     // String literals
  s2 = "Slithey Toves";     // s1 == s2
```

- **The "`+`" operator concatenates strings**
  - Since strings are immutable, this actually creates a new `String`

```
String s = "A string";
s = s + " with appended text";
```

  - Any primitive value or object can be concatenated with a `String`

```
double x = 5.7;
System.out.println("The value of x is " + x);
```

- **`String` methods (see docs for more):**
  - String format(String format_str, Object... args)
  - `int length()`
  - String substring(int start, int end)
  - boolean startsWith(String s)
  - String trim()

# `StringBuffer` and `StringBuilder`

- **`java.lang.StringBuffer` holds a thread-safe mutable sequence of characters**
  - Much more efficient than creating/freeing `String` objects when assembling large strings; use the `append()` method to add content

- **Many editing methods**
  - `reverse()`, `insert()`, `remove()`, etc.
  - Most return the StringBuffer, allowing a fluent programming style

**Example**

```
StringBuffer sb = new StringBuffer("hello");

String s = "world";
int index = 0;

sb.append(s)
  .append("!")
  .setCharAt(0, 'H');

// Find and replace
index = sb.indexOf(s);
sb.replace(index, index + s.length(), "Multiverse");

String s = sb.toString();

System.out.println(s);
```

- **`java.lang.StringBuilder` (Java SE 5+)**
  - Drop-in replacement for `StringBuffer`
  - Not thread-safe, but faster than `StringBuffer` if being accessed by a single thread, which is the usual case

- **Threads will be covered later in the course**

# Module 2

# Arrays

# Arrays

- **An *array* holds a contiguous list of primitive values or object references**
  - All elements must be of the same type

- **Creating arrays**
  - Declare an array  variable

```
// In Java, these are equivalent declarations:
double monthlySales[];
double[] monthlySales;
```

  - Allocate an array with `new`

```
monthlySales = new double[12];  // A year of sales
```

  - By default, all elements of a newly-created array are `null` or 0
  - Array size is fixed once allocated; query using the `length` attribute

```
int numMonths = monthlySales.length;   // no parentheses
```

- **Access array elements with the bracket operator**
  - Indexed starting at zero [0]; index must evaluate to `int` or a type that can be widened to an `int`
  - The bracket operator can be used on either side of an assignment

```

monthlySales[0] = 1500.00;          // Jan
monthlySales[1] = 1200.00;          // Feb
monthlySales[2] = monthlySales[1];  // March same as Feb
...
```

- **Bounds are checked at run time; exceeding the bounds throws `ArrayIndexOutOfBoundsException`**

- **Arrays are always *reference types***

# Arrays of Objects

- **Arrays can hold object *references***
  - Allocating the array only makes room for the references
  - The objects themselves must be created individually

**Example**

```
Circle[] circles;           // array ref
circles = new Circle[10];   // An array of Circle refs

for (int i = 0; i < circles.length; i++) {
  // Create Circle objects
  circles[i] = new Circle( i * .25 );
}
```

- **A Multi-Dimensional Array is an array of array references**

**Example**

```
// Five years of monthly sales
double monthlySalesByYear[][] = new double[5][12];
monthlySalesByYear[0][0] = 1500.00;  // 1st year, Jan
monthlySalesByYear[1][0] = 3000.00;  // 2nd year, Jan
...
monthlySalesByYear[4][11] = 7000.00; // 5th year, Dec
```

- **Rows do not have to be the same length**

**Example**

```
// Four years of daily sales
double dailySalesByYear[][];
dailySalesByYear = new double[4][];     // 2nd dim empty
dailySalesByYear[0] = new double[365];
dailySalesByYear[1] = new double[365];
dailySalesByYear[2] = new double[365];
dailySalesByYear[3] = new double[366];  // Leap year
...
dailySalesByYear[3][59] = 15000.00; // Big sale Feb 29th!
```

# Array Initialization

- **An array reference may be initialized to `null`**

**Example**

```
// No array yet
int[] fibonacci = null;
```

- **Array values may be initialized with a comma-delimited list of expressions in braces**
  - Array length is calculated from value list

**Example**

```
int[] fibonacci1 = new int[] {1, 1, 2, 3, 5, 8, 13, 21};

int[] fibonacci2 = {1, 1, 2, 3, 5, 8, 13, 21};
```

- **An *anonymous* array can be passed as an argument**
  - Array length is calculated from value list

**Example**

```
// This anonymous array is passed as a method parameter

int age = 66;
double shoeSize = 10.5;
String name = "Paul";

System.out.printf("Name: %s, Age: %d, Shoe size: %f",
        new Object[]{name, age, shoeSize});
```

# Copying and Manipulating Arrays

- **Assigning an array copies the reference, not the array**

**Example**

```
int[] src = new int[10];
int[] dest = src;          // Reference to same array
```

- **To copy values between arrays, use the convenience routine `System.arraycopy()`**

**Example**

```
// Allocate destination array
int[] dest = new int[src.length];

// Copy src.length elements into dest
// Starting at src[0], copying to dest[0]

System.arraycopy(src, 0, dest, 0, src.length);
```

  - Questions: what if this was an array of objects? What would be copied?

- **The `java.util.Arrays` class provides `static` methods to help manipulate arrays**
  - search, compare, copy, fill, truncate, sort, convert to a `List` or `String`

**Example**

```
// Automatically allocate destination array
int[] dest2 = Arrays.copyOf(src, src.length);
```

# Enhanced `for` Loop

- **The enhanced `for` loop, sometimes called the for-each loop, is used to traverse arrays and `Collection` classes without an explicit counter**
  - Loop stops automatically after iterating through all elements

**Example**

```
package com.pekia.examples;
public class EchoArguments {
  public static void main(String[] args) {
    for ( String s: args ) System.out.println(s);
  }
}

Output

prompt> java com.pekia.examples.EchoArguments one two three
one
two
three
```

# Arrays as Method Arguments

- **The size of the array is determined by the caller**
  - The enhanced `for` loop makes this easy to process

**Example**

```
package com.pekia.examples;

public class PassingArrays {

  // main() receives an array of Strings

  public static void main( String[] args ){
    for (String s: args ) System.out.println(s);
    int[] a = {25, 26, 27};
    printArray( a );
  }

  // The following method receives an array of ints

  static void printArray( int[] values ){
    System.out.println("An array of ints:");
    for ( int i: values ) System.out.println(i);
  }
}
```

# Varargs

- **An anonymous array can be passed through a variable-length argument list (varargs)**

- **A varargs parameter is declared as** *`<type>... name`*
  - o Must be the last, and only varargs parameter in the list
  - o All values passed by caller must be convertible to the type declared in the parameter list

- **The size of the array is determined by the number of arguments passed by the caller**

**Example**

```
package com.pekia.examples;

public class PassingVarargs {
  // Alternate declaration for main method
  public static void main( String... args ){

    // Pass values as varargs
    printArray("An array of ints:", 25, 26, 27);

    int[] a = {25, 26, 27};
    // Or pass values as array
    printArray("An array of ints:", a);
  }

  static void printArray( String m, int... values ){
    System.out.println(m);
    for ( int i: values ) System.out.println(i);
  }
}
```

# Autoboxing and Varargs

- **If a varargs parameter is declared as type `Object...`, primitive values are autoboxed**

**Example**

```
public class PassingVarargs2 {

  // main gets an array of Strings

  public static void main( String... args ){
    for (String s: args ) System.out.println(s);

    printArray("An array of values:",
                 "twenty-five", 26, 27.0);
  }

  // printArray gets an array of anything

  static void printArray( Object... values ){
    for ( Object o: values ) System.out.println(o);
  }
}
```

- **This feature is used in the formatted output methods**

# Module 3

# Exceptions

# Runtime Exceptions

- **Some programming errors cannot be detected by the compiler**

**Example**

```
public class RuntimeProblems {

  public static void main(String[] args)    {

    int x = 0;       // As far as the compiler knows
    int i = 5;       // these are initialized just fine

    int y = 42 / x;  // integer divide by zero

    int ar[] = new int[3];
    ar[i] = 13;      // bad array index

  }

}
```

- **These result in runtime exceptions**
  - If not handled, program terminates abruptly with a not-too-friendly message

# Traditional Error Handling

- **Traditional (synchronous) error detection involves checking a return value (or some special status variable) after calling a function**

- **Advantages:**
  - Synchronous; prevents further corruption of state after a failed operation

- **Disadvantages:**
  - Inefficient – Checks must be done even if everything goes smoothly
  - Hard to use return value as both status value and a legitimate output value
  - Constructors, initializer blocks, and operators can't be checked this way
  - Compiler does not enforce error checking

- **In many cases, this approach may still be best practice**

# Exceptions

- **Java provides error detection through asynchronous exceptions**

- **An exception object is *thrown* when an error condition is encountered**
  - Exception is *caught* by a block of code (*catch block*) specifically designed to handle the exception
  - If exception is not handled by a local catch block, it propagates up the stack until it is caught
  - An unhandled exception forces a program to exit with an error

- **Advantages**
  - Allows code that works properly to proceed without checks
  - Constructors, initializers, operators, can throw exceptions
  - Compiler can enforce exception handling
  - Allows exception handling to be aggregated in functions higher up the stack

- **Disadvantages**
  - Interrupts the current workflow, and may leave indeterminate state that must be cleaned up
  - Catching exceptions higher up the stack makes it hard to do anything useful to correct them

- **Some libraries use exceptions a lot**
  - I'm looking at you, java.sql

# try-catch-finally

```
public void readFile(String name) {

  FileInputStream fin = null;

  try {
    // FileNotFoundException thrown if file does not exist
    fin = new FileInputStream(name);

    // read file ....

    // IOException thrown if problem closing the file
    fin.close();
  ...
  }
  catch (FileNotFoundException e) {
      System.out.println("Unable to locate file " + name);
  }
  catch (IOException e) {
      System.out.println("File not closed properly");
  }
  finally {
      fin = null;
  }

}
```

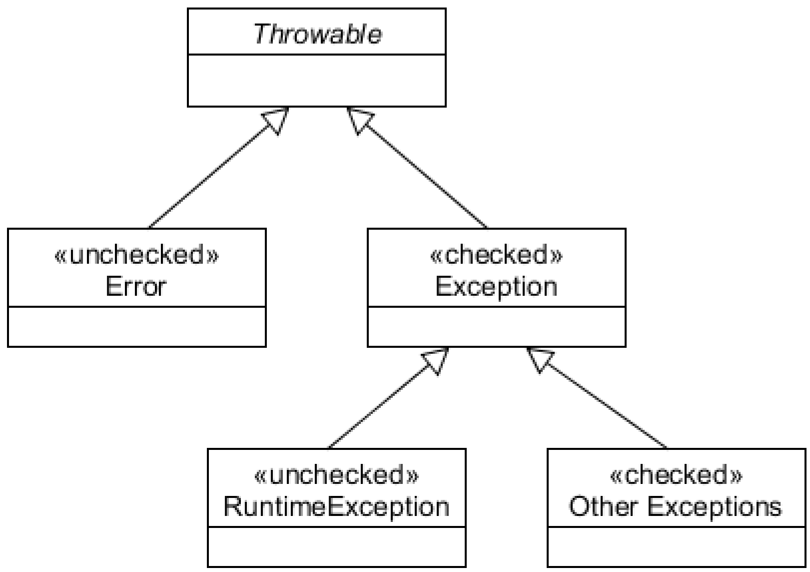
- **Code in a `try` block executes until it completes successfully or an exception is thrown**
  - On an exception, control goes immediately to the appropriate `catch` block
  - The exception is received as an argument to the `catch` block
    - `catch` block can use it to diagnose or report the problem
  - Some code in the `try` block may not be executed!

- **Each `catch` block is specific to a `Throwable` subclass**
  - `catch` blocks must appear in subclass to superclass order

- **Code in the optional `finally` block is *always* executed, either after a successful `try` block or after a triggered `catch` block**
  - Usually used to close streams and clean up before continuing

# Throwable

- **All exception objects must be subclasses of `java.lang.Throwable`, from which they inherit methods, e.g.,**
  - String getMessage()
    - Returns message associated with this `Throwable`
  - void printStackTrace()
    - Prints stack trace to `System.err`




- **Exception types are considered as "Checked" or "Unchecked" by the compiler**
  - If checked exceptions are not handled explicitly, code will not compile
  - Handling is optional for unchecked exceptions; the compiler won't complain
  - *At run time, either will kill your app if you don't handle them*

# Unchecked Exceptions

- **`Error` reports a major JVM error**
  - e.g., running out of memory
  - Impossible or impractical to recover gracefully
  - Best practice: Don't try to catch them

- **`RuntimeException` reports an avoidable logic or data validation error that should have been discovered and corrected during development**
  - e.g., Array index out of bounds exception
  - Best practice: catch and log unchecked exceptions so you can FIX YOUR CODE!!

# Checked Exceptions

- **Other subclasses of `Exception` denote run-time problems that you might anticipate and handle gracefully**
  - e.g., File not found
  - Network connection failure

- **Methods that throw checked exceptions *must* have a `throws` declaration**
  - Warns you that the method *might* throw an exception
  - For unchecked exceptions, a throws declaration is optional

**Example**

```
// Consider the java.io.FileInputStream constructor

public FileInputStream( String fname ) throws FileNotFoundException {
  ...
}
```

- **Code that calls such methods must handle the exception explicitly**
  - You have several options

# Exception Handling Options

- **Catch the exception and try to take some corrective action**
  - You might suggest that the user try something else
- **Catch and rethrow the exception**
  - You could log it for later investigation
- **Catch, then throw a different exception that might be more meaningful farther up the stack**
  - You can attach the original exception if you want
- **Don't catch; hope some other code will catch it**
  - Simply add the exception to your own throws declaration
  - The uncaught exception is passed up the call stack

**Example**

```
public void readFile(String name)
   throws FileNotFoundException, IOException {
  ...
  // FileNotFoundException if file does not exist
  fin = new FileInputStream(name);
  ...
  // IOException if problem closing the file
  fin.close();
  ...
}
```

- **Catch it, ignore it, and hope nothing bad happens**
  - This is ugly but sometimes the only option

# Application-defined Exceptions

- **You can extend the `Exception` class hierarchy to describe application-specific conditions**
  - Extend `Exception` or `RuntimeException` (or a subclass) based on the condition
  - Do not extend `Error`

- **Provide (at least) two constructors**
  - No-parameter constructor
  - A one-parameter constructor that accepts a message

- **Provide fields, setters and getters for other information**
  - The more information returned about the circumstances of the error, the easier debugging will be

**Example**

```
public class FileCloseException extends IOException {

  public FileCloseException() {
    super();
  }

  public FileCloseException(String message) {
    super(message);
  }

}
```

# Throw an Exception

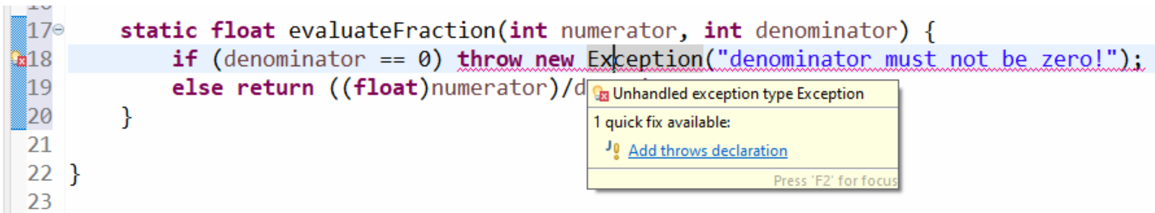
- **When an error is detected, create an instance of the appropriate `Exception` subclass and throw it with the `throw` statement**
  - The exception is passed to the surrounding block
- **Exceptions can be thrown from within `catch` blocks**
  - to rethrow an exception that can't be handled
  - or to substitute an exception that will be more meaningful to the caller
  - The new exception goes to the surrounding caller, not back to the previous `try` block

**Example**

```
public void closeFile (File f)
  throws FileCloseException {

    try {
      f.close();
    }
    catch (IOException e) {
      // throw a more interesting exception
      throw new FileCloseException(
         "File " + f.getName() + " was not closed");
    }
}
```

# Eclipse can Help

- The editor will remind you to add the throws declaration




- And the catch block

# Example

```
package com.example.except;

import java.util.logging.Level;
import java.util.logging.Logger;

public class ExceptionMain {

 static double evalFraction(int num, int den) throws Exception {

     if (den == 0)
         throw new Exception("denominators must not be zero!");

     return ((double)num)/den;
 }


 public static void main(String[] args) {
     int a = 234, b = 0;
     double result = 0.0;

     try {

         result = evalFraction(a, b);
         System.out.println("result is " + result);

     } catch (Exception e) {

         Logger.getGlobal().log( Level.SEVERE,
                     "I can't do math: " + e.getMessage());

     }
 }
}
```

# Section 3-1

# Assertions

# The assert statement

- **The `assert` statement tests a `boolean` expression and throws an `AssertionError` if test is `false`**

**Example**

```
assert (a >= b) : "Error: A less than B";
```

- **Use it to verify that a condition is "as expected" at runtime**
  - Compact syntax is handy when debugging code

- **Pre-condition: check before execution of a code block**

```
private float gravity(float m1, float m2, float dist){
  // Mass must always be positive
  assert m1 > 0 : "m1 " + m1 ;
  assert m2 > 0 : "m2 " + m2 ;
  ...
}
```

- **Post-condition: check after execution of a code block**

```
private float gravity(float m1, float m2, float dist){
  float gravity;
  // calculate gravity
  ...
  // The force of gravity is always positive
  assert gravity >= 0 : "Gravity " + gravity;
  return gravity;
}
```

- **Invariant: check that condition holds during a block**

```
private float gravity(float m1, float m2, float dist){
  float gravity;
  // start calculation
  ...
  assert gravity >= 0 : "Gravity " + gravity;
  // end calculation
  ...
  return gravity;
}
```

# Enabling Assertions

- **Can be enabled/disabled at run time**
  - Allows complex test cases that would cause performance problems if always enabled
- **Programs should run the same whether or not assertions are enabled**
  - Don't use for routine error handling or work that a program requires to function
- **By default, assertions are disabled at run time**
  - The JVM replaces disabled `assert` statements with an empty statement
  - Negligible performance penalty
- **Assertions are selectively enabled/disabled for specific classes**
  - To enable, use the command line switches to the JVM
    `-ea:<argument>` or `-enableassertions:<argument>`
    - argument is the class name, or package with wild card
    - With no argument all assertions are on except system classes
  - To disable
    `-da:<argument>` or `-disableassertions:<argument>`
  - Switches are cumulative

**Example**

```
prompt> java –ea:com.example... com.example.AssertTest
```

- **Assertions in system classes**
  - To enable
    `-esa` or `-enablesystemassertions`
  - To disable
    `-dsa` or `-disablesystemassertions`

# Assertion Example

- **File AssertTest.java**

**Example**

```
// Assertion Example

package mod09.examples;

public class AssertTest {
  public static void main(String args[]){
    assert (args.length != 2 ) : "Two args required!";
  }
}
```

- **Compile and run**

**Example**

```
prompt> javac -d . AssertTest.java

prompt> java –ea mod09.examples.AssertTest

Exception in thread "main" java.lang.AssertionError: Two args required!
 at com.example.AssertTest.main(AssertTest.java:3)
```

# Section 3-2

# Date and Times

# Legacy Date and Calendar

- **java.util.Date**
- **java.util.Calendar**
- **java.util.GregorianCalendar**

# Java SE 8 Date/Time Libraries

- **Replaces earlier `java.util.Date` and `Calendar` classes**
  - But not a drop-in replacement!

- **The date/time library is supported by five packages:**

```
java.time
java.time.chrono
java.time.temporal
java.time.zone
java.time.format
```

- **Satisfies a wide range of use cases for dates and times**
  - Thread-safe, immutable value classes
  - Fully supports ISO-8601 calendar
  - Supports time zones and zone offsets (e.g., changing from standard to DST)
  - Supports alternate chronologies, e.g., Japanese imperial years

- **A "fluent" style of programming**

- **A library to format and parse ISO-8601 dates/times**

# Machine Time

- **Machine view of time is based on the *epoch***
  - Relative to 1970-01-01T00:00:00Z
  - Extends approximately 1 billion years into the past and future
- **An `Instant` represents a discrete point on the timeline**
  - May be positive or negative
  - Stored in nanosecond precision
    - However, your hardware clock will dictate the real accuracy and precision of reported times such as `Instant.now()`
  - Replaces the earlier `java.util.Date` class
    - For compatibility, a `Date` can be converted to/from from an `Instant`
- **A `Duration` represents the amount of time between two instants**
  - May be positive or negative
- **Methods of `Instant` and `Duration` can be used to construct, adjust, compare, query, add, subtract and manipulate times**
  - Since `Instant` and `Duration` objects are immutable, these actually create new instances in many cases

# Instants and Durations

- **Example**

```
import java.time.*;

public class MachineTime {

 public static void main(String[] args) {

  Instant start = Instant.now();

  System.out.printf("Starting at %s%n", start);
  System.out.printf("In the epoch %s%n", Instant.EPOCH);
  System.out.printf("with Max time %s%n", Instant.MIN);
  System.out.printf("and Min time %s%n", Instant.MAX);

  Instant end = Instant.now();
  System.out.printf("Finished at %s%n", end);

  Duration duration = Duration.between(start, end);
  System.out.printf("That took %d milliseconds",
    duration.toMillis());
 }
}
```

# Local Times

- **The human view of time is often context-dependent**
  - It's time for lunch! (today)
  - Let's have breakfast at 8:45 AM (tomorrow)
  - My birthday is March 4th (every year)

- **"Local" times are not associated with a time zone**
  - `LocalTime` represents a time with no date
  - `LocalDate` represents a date with no time
  - `LocalDateTime` represents a date and time

**Example**

```
LocalDateTime now = LocalDateTime.now();
System.out.printf("Now it is %s%n", now);

LocalTime lunch = LocalTime.of(12, 00);
Duration duration = Duration.between(
  now.toLocalTime(), lunch);
long untilLunch = duration.toMinutes();
System.out.printf("%d minutes %s lunch!%n",
  Math.abs(untilLunch), untilLunch<0?"past":"until");

LocalDateTime breakfast =
  now.plusDays(1).withHour(8).withMinute(45);
System.out.printf("... and %d hours until breakfast%n",
  now.until(breakfast, ChronoUnit.HOURS));

LocalDate birthday = LocalDate.of(2015, 3, 4);
System.out.printf("... and %d days until my birthday%n",
  now.toLocalDate().until(birthday, ChronoUnit.DAYS));
```

# Zoned Times

- **A *time zone* is a region of the Earth's surface in which localities share the same standard time**
  - o The default time zone is UTC ("Zero" time)
  - o Each time zone has an offset (in hours and minutes) from UTC
  - o Each time zone has an identifier, e.g., "America/NewYork"

- **A `ZonedDateTime` represents a date and time expressed in a particular ISO-8601 time zone**
  - o Replaces the earlier `java.util.Calendar` class

**Example**

```
// Flight leaves Berlin at 08:45 local time
LocalDateTime depart = LocalDateTime.of(
  2014, Month.JULY, 12, 8, 45, 0);
ZonedDateTime departBerlin = ZonedDateTime.of(
  depart, ZoneId.of("Europe/Berlin"));
System.out.printf(
  "Depart Berlin %s%n", departBerlin);

// Fly for 9 hours
Duration flightTime = Duration.ofHours(9);
System.out.printf("  flight time %s%n", flightTime);

// Arrival time (Berlin)
ZonedDateTime arrive = departBerlin.plus(flightTime);

// What is arrival time in Chicago?
ZonedDateTime arriveChicago = ZonedDateTime.ofInstant(
  arrive.toInstant(),ZoneId.of("America/Chicago"));
System.out.printf("Arrive Chicago %s%n", arriveChicago);
```

# Chronologies

- **Classes and interfaces in the `java.time.chrono` package offer support for developing alternate chronologies**
  - Shipboard time
  - French revolutionary time
  - Lunar calendar
  - Ethiopic/Coptic
  - Japanese imperial years
  - Mayan long count
  - Stardate

# Section 3-3

# Classes and Objects

# Classes

- **A *class* models concepts that cannot be expressed as simple primitive values**
  - Real-world objects – car, person, animal, bank account, etc.
  - Software artifacts – button, window, stack, queue, linked list, etc.
  - A class is defined by a class declaration
  - Inheritance and Polymorphism

**Example**

```
package com.pekia.examples;

public class Shape {
   private int xPosition;
   private int yPosition;

   public Shape(int x, int y) {
     xPosition = x;
     yPosition = y;
   }

   public int getXPosition() { return xPosition; }
   public void setXPosition(int x){xPosition = x;}

   public int getYPosition() { return yPosition; }
   public void setYPosition(int y){yPosition = y;}

   public double getArea() {
     return 0.0;   // default area
   }

   public void draw() {
     System.out.println("Shape: \n" +
      "x = " + xPosition + ", y = " + yPosition );
  }
}
```

# Inheritance

- **Use the `extends` keyword to define a class based on another class**

**Example**

```
public class Circle extends Shape { ... }
```

- **The *subclass* inherits features from its *superclass***
  - Automatically gets all superclass methods and attributes
  - May define additional methods and attributes as needed
  - May redefine (*override*) superclass methods where appropriate

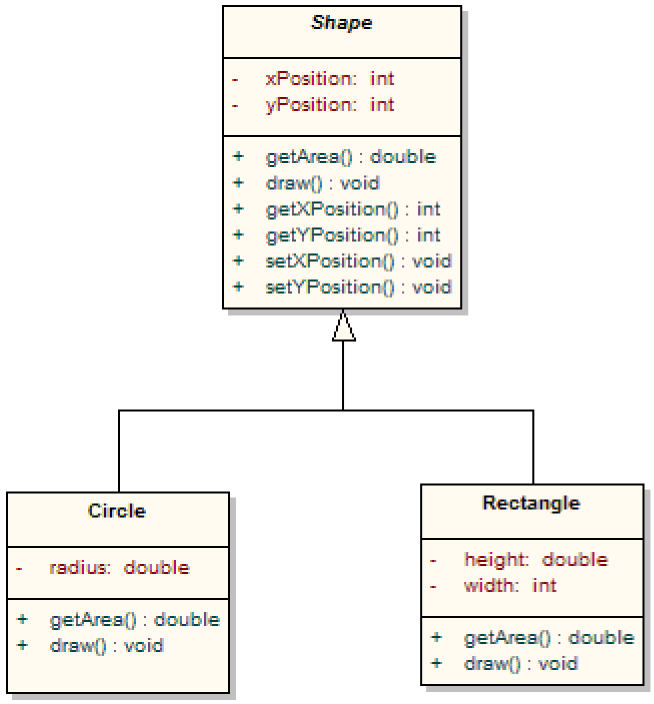
- **An instance of a subclass can always be treated as an instance of its superclass**

**Example**

```
  Shape s = new Circle();    // Perfectly legal
```

# Example: Inheritance

- **The following picture is a UML diagram that shows the subclass relationship**
  - A `Circle` or a `Rectangle` can always be treated as a `Shape`



# Accessing Superclass Constructors

- **Constructors are only responsible for their own class**

- **A subclass should call its superclass constructor to initialize inherited variables**

**Example**

```
package com.pekia.examples;

public class Circle extends Shape {
  private double radius;

  ...
  public Circle(int x, int y, double r) {
    super(x, y);
    setRadius(r);
  }
  ...
}
```

  - `super()` must be the first statement in the constructor
  - May pass arguments

- **If superclass constructor is *not* called explicitly, compiler automatically inserts a call to `super()` with no args**
  - In this example, that would cause a compile time error because `Shape` does not have a no-arg constructor

# Overriding Methods

- **Inherited methods may be *overridden* in subclasses**
    - `Circle` provides implementations of `draw()` and `getArea()` that are appropriate for a circle

**Example**

```
...
public class Circle extends Shape {
  ...
  // overridden methods
  public void draw() {
    System.out.println("Circle at x="
        + getXPosition()      // Call superclass
        + ", y="              // methods; variables
        + getYPosition() );   // are private
  }
  public double getArea() {
    return Math.PI * radius * radius;
  }
}
```

- **The appropriate method is *always* invoked based on the *actual* type of an object, *not* the type of its reference variable**

**Example**

```
  Shape s = new Circle();
  double a = s.getArea();   // getArea() from Circle
```

# `protected` Members

- **Tradeoffs are made between efficiency and encapsulation**
- **`Circle` could be more efficient if `Shape` grants access to its `private` data members:**

**Example**

```
public class Shape {
  protected int xPosition;
  protected int yPosition;
  ...
}
```

**Example**

```
public class Circle extends Shape {
  ...
  public void draw() {
    System.out.println("Circle at x="
           + xPosition      // Use protected
           + ", y="         // members directly;
           + yPosition );   // Save a function call
  }
  ...
}
```

- **The programmer of `Circle` is now dependent on `Shape` internal structure**
  - This might cause problems later if `Shape` is modified
  - This is an architectural decision that must be considered carefully when designing classes

# Accessing Superclass Members

- **The `super` keyword can be used to access a superclass member (variable or method) that is hidden by a member of the subclass**

- **`Circle` could be more efficient by accessing `Shape's` version of `draw()`**

**Example**

```
public class Circle extends Shape {
  ...
  public void draw() {
    super.draw();
    System.out.println("  Circle, r = " + radius);
  }
  ...
}
```

  - This could save the `Circle` programmer some work, and avoids duplication of code to deal with `xPosition` and `yPosition`

- **The programmer of `Circle` is now dependent on `Shape` behavior**
  - This is another architectural decision that must be considered carefully during design

# `abstract` Classes and Methods

- **The `abstract` keyword marks methods or classes that are somehow incomplete**

- **Abstract classes cannot be instantiated, and *must* be subclassed**

**Example**

```
public abstract class Shape { ... }
```

  - Abstract classes may have abstract methods

- **Abstract methods *must* be overridden by subclasses**

**Example**

```
public abstract double getArea(); // no implementation!
```

  - Abstract methods cannot have a method body
  - Subclasses are considered abstract until <u>every</u> abstract method has been overridden with a concrete implementation

# **Abstract** `Shape` **Class**

**Example**

```
package com.pekia.examples;

public abstract class Shape {
   private int xPosition;
   private int yPosition;

   protected Shape(int x, int y) {
     xPosition = x;
     yPosition = y;
   }

   public int getXPosition() {
     return xPosition;
   }

   public void setXPosition(int x) {
     xPosition = x;
   }

   public int getYPosition() {
     return yPosition;
   }

   public void setYPosition(int y) {
     yPosition = y;
   }

// Design decision: no implementation of these
// methods in the Shape superclass
// Subclasses must override them

   public abstract double getArea();
   public abstract void draw();

}
```

# Polymorphism

- **A `Picture` holds an array of `Shapes`**
  - To draw the picture, draw the shapes one by one
  - `Picture` doesn't need to know about specific subclasses of `Shape`
  - New `Shape` subclasses can be added without modifying, recompiling or retesting `Picture`

```
package com.pekia.examples;

public class Picture {
   private int count = 0;

   // Create an array of Shape references
   // Each of these references could refer
   // to any Shape subclass

   private Shape[] shapes = new Shape[10];

   // Any subclass of Shape can be added

   public void add(Shape s) {
     shapes[count++] = s;
   }

   public void draw() {
     for (int i = 0; i < count; i++){
     shapes[i].draw();    // polymorphic call
   }
}
```

# Using the `Picture` Class

```
// File: Rectangle.java - another Shape subclass

package com.pekia.examples;
public class Rectangle extends Shape {
   private double width, length;

   public Rectangle(int x,int y,double w,double l) {
     super(x, y);        // in Shape
     width = w;
     length = l;
   }
   public double getArea(){ return width * length; }
   public void draw() {
     System.out.println("Rectangle");
   }
}

// File: TestPicture.java

package com.pekia.examples;
public class TestPicture {
   public static void main(String[] args) {
     Picture p = new Picture();

     p.add(new Circle(1, 1, 1.0));
     p.add(new Rectangle(1, 1, 1.0, 1.0));
     p.add(new Rectangle(2, 2, 2.0, 2.0));
     p.add(new Rectangle(3, 3, 3.0, 3.0));
     p.add(new Circle(2, 2, 2.0));

     p.draw();
   }
}

Output:

Circle at x=1, y=1
Rectangle
Rectangle
Rectangle
Circle at x=2, y=2
```

# `final` Classes and Methods

- A **`final`** class *cannot* be subclassed

**Example**

```
package com.pekia.examples;

// Don't allow subclasses of Square
public final class Square extends Shape {
   private double side;

   public Square(int x,int y,double side) {
     super(x, y);        // in Shape
     this.side = side;
   }
   public double getArea(){ return side * side; }
   public void draw() {
     System.out.println("Square"); }
}
```

- A **`final`** method *cannot* be overridden

**Example**

```
package com.pekia.examples;

public class Circle extends Shape {
  ...
  //  Subclassing may be OK, but
  //  There's no alternative for the area of a circle!
  public final double getArea() {
    return Math.PI * radius * radius;
  }
  ...
}
```

# The `Object` Class

- **Every Java class implicitly extends `java.lang.Object`**
  - An instance of any class is an `Object`

- **At some level, all objects can be treated alike**
  - Put in collections, compared, printed

- **Methods of `Object` are available in all objects; some can be overridden:**
  - `public boolean equals(Object o)`
    - Default returns `true` if invoking object has the same reference as o; subclasses should override this method to compare internal values
  - `public String toString()`
    - Returns a `String` representation of the object, by default:

```
"ClassName@hex_hashcode"
```

    - Subclasses may override to produce a textual representation of the object state; e.g., `Date` objects return a string like this:

```
"Thu Feb 22 06:33:14 EST 2007"
```

  - `public final Class getClass()`
    - Returns an object that describes the class of the object, including variables, methods, etc. Can be used to call the methods of the object.
    - Allows code to interact with objects that were not defined at compile time
  - `protected Object clone()`
    - Returns an equivalent copy of the current object. Default performs "shallow" copy by performing member-by-member assignment

# Section 3-4

# Object-Oriented Programming

# Example: The `Circle` Class

- **We'll improve upon this basic design:**

**Example**

```
package com.pekia.examples;

public class Circle {

  // Attributes

  int xPosition;
  int yPosition;
  int radius;

  // Methods

  void draw() {
    // Code to draw a circle ...
    System.out.println("Circle at x=" +
      xPosition + ", y=" + yPosition );
  }

  double getArea() {
    return Math.PI * radius * radius;
  }
}
```

# Accessing Members

- **Attributes and methods are accessed with dot (`.`) notation**
  - By default, classes in the same package can access each others' members

**Example**

```
package com.pekia.examples;

public class TestCircle {

  public static void main(String[] args) {
    // create a reference on the stack
    Circle c;
    // Allocate a Circle object in the heap
    c = new Circle();

    // Set circle instance variables
    c.radius = 5;
    c.xPosition = 10;
    c.yPosition = 10;

    // call the draw method
    c.draw();

    // call the getArea() method
    double a = c.getArea();
    System.out.println("The area of c is: " + a);
  }
}
```

# Encapsulation

- **Use *access control modifiers* to hide variables or methods**
  - Compile-time errors will flag improper usage
- **Define *setter/getter methods* that access data values while hiding implementation details**
  - Methods may check for bad values, convert to/from internal formats. etc.
- **Provide *constructors* to initialize object state**

# Class Initialization

- **When a class is loaded into the JVM, its `static` variables are initialized**
  - If not initialized explicitly, are automatically set to `0`, `null` or `false`, depending on type
  - Explicit initialization can be done by assigment where a `static` variable is declared

    **Example**

    ```
    public static final double MAX_RADIUS = 20.0;
    public static final double MAX_CIRCUM =
                        2.0 * MAX_RADIUS * Math.PI;
    ```

  - Expression are OK as long as operands are initialized first

- **A `static` initialization block can be used when initialization involves loops, arrays or other complex code**

  **Example**

  ```
  public class MathematicalSeries {
    public static final long fibonacci[] = new long[75];
    static {
      fibonacci[0] = fibonacci[1] = 1;
      for (int i=2; i<fibonacci.length; i++)
        fibonacci[i] = fibonacci[i-1] + fibonacci[i-2];
    }
  }
  ```

  - Executed once, when class is loaded
  - If multiple `static` blocks, they are executed in order

# Object Initialization

- **When an object is created, its instance variables are initialized**
  - If not initialized explicitly, are automatically set to `0`, `null` or `false`, depending on type
  - Explicit initialization can be done by assignment where an instance variable is declared

```
private double radius = 1.0;
```

- **The *constructor* is a special method that is invoked by the `new` operator to dynamically initialize an object**
  - Has the same name as the class, but no return type
  - May accept zero or more parameters
    - Overloaded constructors may be defined with distinct parameter lists
    - The "default" constructor takes no parameters
    - The compiler provides a default constructor automatically *only if no* constructors are coded by the programmer

- **The constructor must ensure that the overall state of the new object is suitable for its intended use**
  - Can be used to limit the ways in which an object can be created, or enforce initialization in particular ways
  - May compute values for instance variables and create dependent objects

# Constructors

**Example**

```
package com.pekia.examples;

public class Circle {

// Use setXxx() methods for value checking

  public Circle() {
     setXPosition(0); // Default values
     setYPosition(0);
     setRadius(1.0);
  }
```

1:

```
  public Circle(int x, int y, double r) {
     setXPosition(x);
     setYPosition(y);
     setRadius(r);
  }
...
  // Design decision:
  // The radius can only be set at creation time
  // Since other classes cannot call this method
  // it can be modified without impacting them

  private void setRadius(double r) {
     radius = ( r < 0.0 ) ? 0.0 : r;
  }
}
```

**Example**

```
// This DOES change the programming interface
// Now, Circles initialize themselves

Circle c1 = new Circle();              // OK
Circle c2 = new Circle(10, 10, 15.0);  // OK

c2.setRadius(10.0);                    // does not compile
```

# Initializers and Finalizers

- **Instance variables can be initialized in an unnamed *initialization block***

**Example**

```
public class Circle {
   {
     // initializer code
   }
}
```

  - Copied into all constructors by the compiler

- **The *finalizer* is called by the JVM after an object is marked for garbage collection**

**Example**

```
protected void finalize() throws Throwable {
     // do cleanup here
    super.finalize();
}
```

  - Do not use! Since the garbage collector is asynchronous, there is no way to know when `finalize` method will be invoked.
    - If program exits before object is garbage collected, `finalize()` is never called
  - Classes that need to release resources synchronously should provide a method to do so ( e.g., `close()`, `remove()`, `destroy()`, etc. )

# `this`

- **Within an instance method, the keyword `this` holds a reference to the current object, which can be used for several purposes:**

- **Call overloaded constructors**

```
this( args );
```

  - Can only be called from another constructor, and must be first statement in the constructor

- **Distinguish between a member variable and a local variable of the same name**

```
public class Circle {
  private int xPosition;
  ...
  public void setXPosition( int xPosition ){
    this.xPosition = xPosition;
  }
  ...
}
```

  - Some programmers prefer to adopt naming conventions to avoid this usage

- **Pass the current object as an argument to another method**

```
System.out.println(this);
```

- **Return a reference to the current object**

```
return this;
```

# `this` Example

**Example**

```
package com.pekia.examples;

public class Circle {
   private int xPosition;
   private int yPosition;
   private double radius;

   public Circle() {  // Why duplicate code when
     this(0, 0, 1.0); // it is there for the taking?
   }

   public Circle(int x, int y, double r) {
     setXPosition(x);
     setYPosition(y);
     setRadius(r);
   }
   ...
   private void setRadius(double radius) {
     this.radius = ( radius < 0.0 ) ? 0.0 : radius;
   }
   ...
}
```

# Section 3-5

# Nested Classes

# Nested Classes

- **A *nested* class is declared inside another class**
  - Compiler generates the class file `Outer$Nested.class`

```
public class Outer {
  public static class Nested {
  }
}
```

- **A `static` nested class is just a packaging technique**
  - Allows several `public` classes to be defined in the same file
  - Effectively, both classes act like top-level classes

```
  Outer o = new Outer();
  Outer.Nested n = new Outer.Nested();
```

- **A non-`static` nested class is called an *inner* class**
  - An instance of the inner class can exist *only* within an instance of the outer class
  - Inner classes and their outer class have access to each others' `private` instance variables

- **Inner classes are used to encapsulate functionality that needs efficient, tightly-coupled access to a larger context**
  - Event listeners in graphical user interfaces, iterators and links in collection classes, etc.

# `private` Inner Class

- **A `private` inner class cannot be accessed by other top-level classes, and is useful only to its surrounding class**

**Example**

```
package com.pekia.examples;

public class LinkedStack  {
    private Link head = null;

    public LinkedStack push(Object obj) {
        head = new Link(obj, head);
        return this;    // allow chained push
    }
    public Object pop() {
        if (head == null) return null;
        Object obj = head.data;   // access inner
        head = head.next;         // class members
        return obj;
    }

    private class Link {
      private Link next;
      private Object data;
      private Link(Object data, Link next){
        this.data = data;
        this.next = next;
      }
    }
    ...
```

- The outer class `LinkedStack` represents a stack, and supports `public` methods to `push()` and `pop()` objects
- `private` inner class `Link` implements the list, and is only used internally; it is never visible to external classes

# `public` Inner Class

- **A `public` inner class can provide controlled access to `private` data kept by an outer class**
  - `Iterator` uses `private` fields in outer class and `Link`

**Example**

```
// LinkedStack - continued
 ...
   public Iterator iterator() { // factory method
     return new Iterator();     // accesses private
   }                            // contructor

   public class Iterator {
     private Link current;
     private Iterator() {
       current = head;          // access outer
     }
     public Object next() {
       if (current == null) return null;
       Object obj = current.data;  // access Link
       current = current.next;     // inner class
       return obj;
     }
   }
}
```

- **Other top-level classes can refer to `public` inner classes by their nested class names**
  - Inner class `Iterator` has the fully-qualified name `com.pekia.examples.LinkedStack.Iterator`
  - `Iterator` constructor is `private`, so outer class provides a `public` factory method for the benefit of other classes

# JavaBeans

- **JavaBeans are Plain Old Java Objects (POJOs)**
  - Not to be confused with the "Enterprise Java Beans" covered in the advanced course

- **A bean follows naming and coding conventions that make it easy to use in applications**
  - A `public` no-argument constructor
  - Implements the `Serializable` or `Externalizable` interface (discussed in module 10)
  - Provides `public` set and get methods that are used to manipulate at least some of its features
    - Such features are called *properties*
  - Methods follow naming conventions that make them accessible through the Java reflection interfaces
    - Developers can integrate these objects into applications without changing/recompiling *application* code

- **The predominant use of beans is as data objects**
  - They carry a set of data values
  - They can be passed easily between tiers of an application

# JavaBean Example

- **A *data bean* encapsulates application data**
  - Often used used as transfer objects to carry packets of information between components

**Example**

```
package com.pekia.examples;
import java.io.*;

public class NameBean implements Serializable {
 // Internal variable names are not important
 private String lName="", fName="";

 public NameBean() { }
 public NameBean(String first, String last) {
    lName = last;
    fName = first;
 }
 // Method names imply the property names
 public void setFirstName(String n) {
   fName = n;
 }
 public String getFirstName() {
   return fName;
 }
 public void setLastName(String n) {
   lName = n;
 }
 public String getLastName() {
   return lName;
 }
 public String getName() {
   return lName + ", " + fName;
 }
}
```

  - This bean has the properties `firstName`, `lastName` and `name`

# Section 3-6

# Interfaces

# Interfaces

- **Interfaces are like pure abstract classes**
  - Provide form but no implementation

- **Declared using `interface` keyword**
  - Like a class, a `public` interface goes in its own file

**Example**

```
// File: Drawable.java

package com.pekia.examples;

public interface Drawable {
    void draw();
}
```

  - `Shape` no longer needs to declare this method

- **All methods in an interface are declared without implementations**
  - Methods are implicitly `public abstract`

- **May define constants**
  - Any member variables are implicitly `public static final`, and must be initialized where declared
  - No `static` blocks

# Interfaces and inheritance

- **Interfaces can *extend* other interfaces**
- **Classes can *implement* interfaces**

```
public class Circle implements Drawable {
}
```

  - This is a type of inheritance
  - If the implementing class does not supply an implementation for each inherited method, it will be `abstract`

- **Classes can extend only one superclass, but may implement multiple interfaces**

**Example**

```
public class Circle extends Shape
    implements Drawable, Serializable {

    public void draw() {
        System.out.println("Circle");
    }
    ...
}
```

- **Some interfaces (e.g., `Serializable`) are marker interfaces**
  - They declare no methods, but indicate that the implementing class can be treated in a certain way

# Revised `Picture` Class

- **Maintains a list of `Drawable` references instead of `Shape` references**

**Example**

```
package com.pekia.examples;

public class Picture {
  private int count = 0;

 // Picture class can manage any kind of Drawable,
 // whether it is a Shape or not

  private Drawable[] drawables = new Drawable[10];

 // Now any kind of Drawable can be
 // added to the picture

  public void add(Drawable d) {
    drawables[count++] = d;
  }

  public void draw() {
    for (int i = 0; i < count; i++){
       drawables[i].draw();     // polymorphism
    }
  }
}
```

- **Picture does not need to know any other details about the things that it draws**

# Casting Object References

- **Casting to a superclass type (*upcasting*) is automatic**

```
Shape s = new Circle();
```

- **Casting to a subclass (*downcasting*) is permissible only if the object is of a compatible type**

```
Circle c = (Circle)s;
```

  - Requires the cast operator
  - Throws a `ClassCastException` if the object referenced by `s` is not a subclass of `Circle`

- **The `instanceof` operator can be used to check the class of an object**
  - Returns `true` if a cast could be performed without throwing an exception, but overuse leads to badly-designed code like this:

```
public class Picture {
  ...
  public void draw() {
    for (int i = 0; i < count; i++){
      if ( drawables[i] instanceof Circle ){
         // code to draw a circle
      } else if ( drawables[i] instanceof Square ){
         // code to draw a square
      } else if ( drawables[i] instanceof Rectangle ){
         // code to draw a rectangle
      } else {
         // The pain goes on and on
      }
    }
  }
```

# The `Comparable` Interface

- **The core libraries have many interfaces that can be implemented**
  - In the on-line documentation, interface names are in italics

- **Example: objects implementing `Comparable` can be sorted**
  - Must provide a single method named `compareTo()`
  - Return value indicates ordering; negative if `this` is less than `obj`, zero if equal, positive if greater

**Example**

```
public class Circle extends Shape
    implements Comparable, Drawable {
  ...
  public int compareTo(Object obj) {
    Circle a = (Circle) obj;   // cast obj to Circle
    return (int) (this.radius - a.radius);
  }
}
```

- **Objects in an array are sorted with `Arrays.sort()`**

**Example**

```
import java.util.*;
...
Circle[] circles = new Circle[10];
for (Circle c: circles) {
   c = new Circle(0, 0, Math.random());
}
Arrays.sort(circles);
```

# Section 3-7

# Generics

# Generics

- **Using generics, it is possible to create a single class that works with different data types**
  - The type on which it operates is specified at compile time as a *type parameter*

- **A generic class is declared with one or more *type parameters*, enclosed in angle brackets**

**Example**

```
public class NumberHolder<T> { ... }
```

  - The type parameter **T** is a placeholder for the actual data type used within the class
    - By convention, a type parameter name is a single uppercase letter
    - Read as "`NumberHolder` using type `T`"

- **A type parameter value *must be a class or interface type***
  - Primitive data types are not allowed

- **A comma-separated list of parameters is permitted**

**Example**

```
public class VeryGenericClass<T, U, V> { ... }
```

- **The diamond operator <> infers creation types from context**

**Example**

```
// For each unique name, a list of nicknames
Map<String, List<String>> nicknames = new HashMap<>();
```

# Type Parameters

- **In implementation, the type parameter name is used wherever the data type that it represents would be used**
  - Declare local or member variables
  - Declare method parameter data types
  - Declare method return types

**Example**

```
// NumberHolder defined as a generic class

package com.pekia.examples;

public class NumberHolder<T> {

  // Parameter used as member variable type
  private T value;

  // Parameter used as constructor argument type
  public NumberHolder(T value) {
    this.value = value;
  }

  // Parameter used as method argument type
  public void setValue(T value) {
    this.value = value;
  }

  // Parameter used as return type
  public T getValue() {
    return value;
  }
}
```

# Using a Generic Class

- **Type parameters are provided by the code that uses a generic class**

**Example**

```
package com.pekia.examples;

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

   // Create a NumberHolder using type Double
   Double d1 = new Double(123.4);

    NumberHolder<Double> nh
      = new NumberHolder<Double>(d1);

    // Retrieve the object reference held by the class
    // No casting required!
    Double d2 = nh.getValue();

    // Change the value
    nh.setValue(new Double(567.8));

    // Autoboxing works as well
    nh.setValue(432.1);

    // However, this error won't compile
    Integer i = (Integer)nh.getValue();

    // Neither does this one
    nh.setValue(new Byte(125));
  }
}
```

- **Runtime errors are prevented by the compiler**
  - `NumberHolder<Double>` means "`Double` objects only!"

# Bounded Parameter Types

- **`NumberHolder` (thus far) puts no restrictions on the data type that could be used as a parameter**

```
// This is OK
NumberHolder<Double> nhd =
     new NumberHolder<Double>(123.4);

// This is OK too?  It's not really a number...
NumberHolder<String> nhs =
     new NumberHolder<String>("Hello World");
```

- **Parameter types can be restricted (bounded) by using the `extends` keyword**

```
// Restrict the class parameter to a subclass of the
// Number class

public class NumberHolder<T extends Number> { ... }
```

- **Compiler now ensures that parameter type is a subclass of `Number`**

```
// This does not compile
NumberHolder<String> nhs =
    new NumberHolder<String>("Hello World");
```

# Type Erasure

- **Generics are a compile-time construct**
  - No additional bytecode is actually added to the class file
  - In implementation code, the "actual" data type of should be assumed to be the bounding type, or Object

**Example**

```
// Use the same class in three different forms
NumberHolder<Double> nhDouble =
       new NumberHolder<>(123.4);

NumberHolder<Integer> nhInteger =
       new NumberHolder<>(567);

// "raw" type
NumberHolder nhRaw = new NumberHolder(12);
```

- **Generic and non-generic usages of `NumberHolder` use the same class file**
  - Supports backwards compatibility for legacy code
  - Avoids generating extra class files for type specializations
  - However, using the "raw" type generates a compile time warning

# Module 4

# The Collections Framework

# Collections

- **A container is an object that holds a collection of other objects**
  - Objects can be inserted, accessed and removed
  - The container may sort, organize, or provide alternate ways of accessing the objects

- **A container reference may be passed as an argument or return value**
  - A convenient way to ship groups of data between methods

- **An array is a simple container, with significant limitations**
  - Size is fixed
  - All members values must be the same type
  - Methods and attributes cannot be added to an array
  - Can't be used as a superclass

- **The Collections framework provides a set of generic collections**
  - in java.util

# A Better Picture

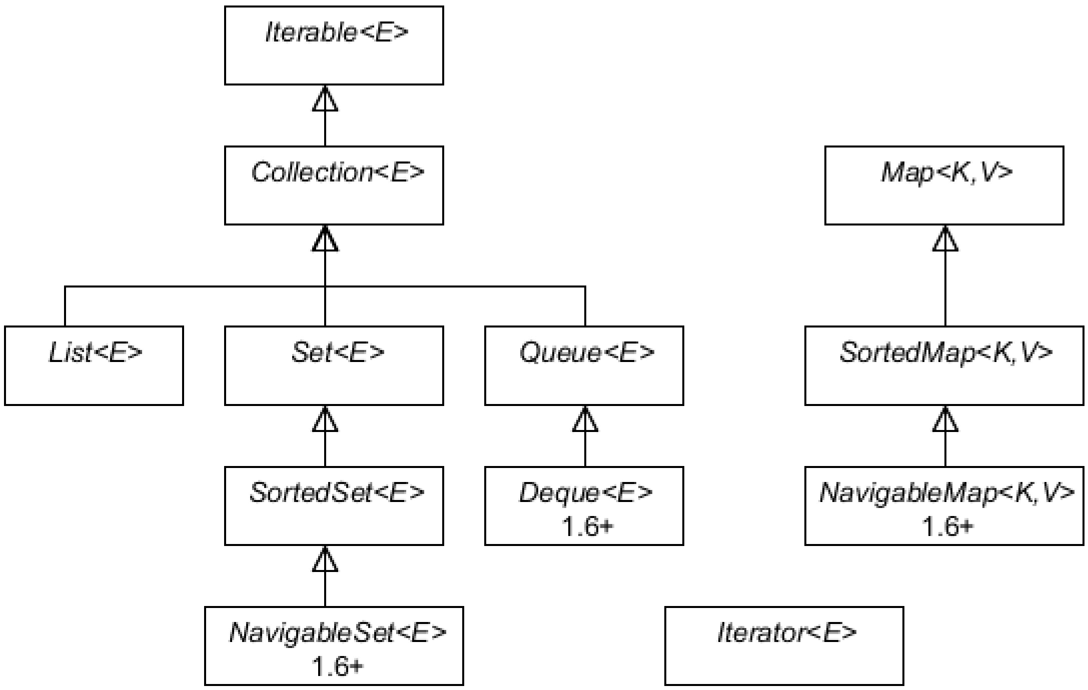
- **The Collection classes can be used to fix a problem from an earlier module**

**Example**

```
import java.util.List;
import java.util.Vector;

public class Picture {

  private List<Drawable> drawables =
        new Vector<Drawable>();

  // Now we can add any number of Drawable objects!
  public void add(Drawable d) {
    drawables.add(d);
  }

  public void draw() {
    // Traverse with enhanced for loop
    for (Drawable d: drawables) d.draw();
  }
}
```

- **This `Picture` is type-safe, doesn't overflow the array, and has only three lines of executable code**

# Collections Interfaces




- **Each interface may have multiple implementations**
  - Classes implementing the `Collection<E>` interface hold collections of single objects
  - Classes implementing the `Map<K,V>` interface hold collections of key/value object pairs
  - Classes implementing the `Iterator<E>` interface are used to traverse objects in containers

# The `Collection<E>` Interface

- **Represents a group of elements with no requirements or constraints**
  - Declares signatures for methods common to all containers:
    - `add(E e)` adds the specified element (optional)
    - `remove(Object o)` removes an instance of the specified element, if present (optional)
    - `clear()` removes all elements (optional)
    - `int size()` returns the number of elements in the collection
    - `isEmpty()` returns true if this collection contains no elements
    - `contains(Object o)` returns true if collection contains this element
    - `iterator()` returns an iterator
    - `toArray()` returns an array containing all elements

- **Methods that add or remove elements are described as "optional" in the specification**
  - This feature allows the development of immutable container subclasses
  - Unsupported methods throw exceptions if invoked

# `Collection<E>` Subinterfaces

- **A class implementing `List<E>` guarantees that its elements are maintained in insertion order**
  - The user can insert and access elements using an index
  - Duplicate elements are allowed

- **A class implementing `Queue<E>` maintains its elements as a First-In, First-Out sequence**
  - Elements are always inserted at the tail of the queue, and removed from the head
  - Index-based access to the elements is not allowed
  - Duplicate elements are allowed

- **A class implementing `Set<E>` does not permit duplicate elements**
  - `add()` returns `false` if element is already in the collection
  - A class implementing `SortedSet<E>` guarantees that elements can be traversed in sorted order
    - Elements must implement the `Comparable<T>` interface, or
    - The container must be associated with a *comparator* object which defines how the elements in the container are to be sorted
  - A class implementing `NavigableSet<E>` can report the closest match for given search targets

# `Map<K,V>` Interfaces

- **A class implementing `Map<K,V>` stores elements as key-value pairs**
  - Keys must be unique
    - In general, immutable objects (like `Integer` or `String`) should be used as keys, but this is not enforced
  - Duplicate *values* are allowed
  - Element sequence is not necessarily maintained

- **A class implementing `SortedMap<K,V>` guarantees that its elements (key-value pairs) can be traversed in a specified key-based order**
  - Ordering is specified one of two ways
    - The keys must implement the `Comparable` interface, or
    - A `Comparator` object must be associated with the container
  - A class implementing `NavigableMap<K,V>` can report the closest match for given search targets

# Collection Implementations

| | Interface | | | |
|---|---|---|---|---|
| **Implementation** | **Set<E>** | **List<E>** | **Deque<E>** | **Map<K,V>** |
| **Hash Table** | HashSet | | | HashMap<br>Hashtable |
| **Resizable Array** | | ArrayList<br>Vector | ArrayDeque | |
| **Balanced Tree** | TreeSet | | | TreeMap |
| **Linked List** | | LinkedList | LinkedList | |
| **Hash Table + Linked List** | LinkedHashSet | | | LinkedHashMap |

- **Concrete classes implement the interfaces**
  - Have names of the form "Implementation Style + Interface Name"

- **Implementations are distinguished by their performance characteristics**
  - Fast insertion and removal of elements: linked-list implementation
  - Index-based access: array-based implementation
  - Access times independent of container size: hashtable implementation

- **Interface and implementation can be chosen based on algorithmic needs (i.e., queue versus sorted list) and performance requirements**

# Code to the Interface

- **Best practice: assign a collection to a reference of an *interface* type rather than the implementation *class* type**
  - Code to the interface, not the implementation
    - Alternate implementations can be substituted without affecting the rest of the code

**Example**

```
List<Integer> myList = new ArrayList<Integer>();

// Changing implementation
// myList has same API; has no impact on client code

List<Integer> myList = new LinkedList<Integer>();
```

- **References *can* be cast to a concrete type to access implementation-specific methods**
  - This is required only when tuning the behavior of a collection implementation
  - May make it difficult or impossible to exchange classes later on

# `List<E>` Implementations

- **Classes that implement the `List<E>` interface allow access to elements by index**
  - `ArrayList<E>`
    - Uses resizable array for element storage; best all-round `List<>` impl
    - Allows the programmer to specify an initial capacity and change it if necessary during the application life
  - `LinkedList<E>`
    - Implements both `List<E>` and `Deque<E>` interfaces
    - Doubly-linked list implementation provides more efficient `add()` and `remove()` methods than other implementations
    - Extensive and efficient addition and removal of elements

**Example**

```
// An array of String obects
String names[] = {"Bob", "Will", "Dennis" };

// Create an ArrayList
List<String> namesList = new ArrayList<String>();

// Add array contents into the ArrayList
namesList.addAll(Arrays.asList(names));
// Add another element
namesList.add("George");

// Does namesList contain the String "Bob"?
int b = namesList.indexOf("Bob");
System.out.println(
    b > -1 ? "Hello Bob" : "Bob Not found" );

// What element comes after Bob?
String s = namesList.get(b+1);
```

# `Set<E>` implementations

- **Classes that implement the `Set<E>` interface discard duplicate elements**

- **`HashSet<E>`**
  - No guarantee of element order or sequence, storage order will probably be different than the insertion order
  - Initial capacity can be specified at creation time
  - Insertion and retrieval times are independent of the size of the container

- **`LinkedHashSet<E>`**
  - A hashing mechanism promotes fast insertion and removal of elements with no duplicates
  - A doubly-linked list maintains the insertion order
  - Efficient element insertion and removal is independent of container size

- **`TreeSet<E>`**
  - Implements the `SortedSet<E>` interface
  - Elements are maintained in sorted order
  - Elements must implement the `Comparable` interface, or a comparator object must be associated with the container

# `Queue<E>` implementations

- **Classes that implement the `Queue<E>` interface are used to move objects between subsystems and/or threads**
  - Queues may reorder or filter elements

- **`LinkedList<E>`**
  - Implements both `List<E>` and `Queue<E>` interfaces
  - Extensive and efficient addition and removal of elements

- **`PriorityQueue<E>`**
  - Elements with a higher priority are sorted into positions nearer the head
  - FIFO order is maintained for elements with the same priority
  - Elements must implement the `Comparable` interface, or a comparator object must be associated with the container

- **Other `Queue<E>` implementation classes are found in the `java.util.concurrent` package**
  - Designed for use with multi-threaded applications

- **See documentation for specific FIFO needs**

# `Map<K,V>` Implementations

- **These classes implement the `Map<K,V>` interface**
  - Elements are stored as key-value pairs
  - Duplicate values OK, but duplicate keys are not allowed

- **`HashMap<K,V>`**
  - A hash algorithm calculates the storage position of each element
  - Element insertion sequence and sort order are not maintained
  - Insertion and removal times do not vary with container size

- **`LinkedHashMap<K,V>`**
  - Keys are maintained with a doubly-linked list
  - Element insertion sequence is maintained

- **`TreeMap<K,V>`**
  - Implements the `SortedMap<K,V>` interface
  - Elements are maintained in order, sorted by key
  - Key types must implement the `Comparable` interface, or a comparator must be associated with the container
  - If order is not important, it is more efficient to use the `HashMap<K,V>` class

# The `Iterator<E>` Interface

- **All collection classes provide a method named `iterator()`**
  - Returns an object that implements the `Iterator<E>` interface

- **Iterator methods are used to traverse the elements of a container**
  - `hasNext()` returns `true` if there are remaining elements
  - `next()` returns the next element as type `E`
    - Order is determined by the underlying implementation
  - `remove()` removes the last element returned by the `Iterator`
    - Only one `remove()` call is allowed per `next()` call

- **Collections that implement the `List<E>` interface also provide a method named `listIterator()`**
  - Returns an object which implements the `ListIterator<E>` interface
  - Allow bidirectional movement through the collection via the methods `next()` and `previous()`

# `ListIterator<E>` Example

**Example**

```
List<Integer> myList = new ArrayList<Integer>();

// Add four autoboxed elements to the collection
myList.add(23);
myList.add(45);
myList.add(149);
myList.add(25);

// Obtain a ListIterator object positioned at the
// last element in the collection
ListIterator<Integer> it =
       myList.listIterator(myList.size());

// Use the iterator to step backwards through the
// collection
while (it.hasPrevious()) {
    Integer myInt = it.previous();
    System.out.println(myInt);
}

// Obtain a new ListIterator object positioned at
// the beginning of the collection
it = myList.listIterator();

// Remove first three elements of the collection
for (int i = 0; i < 3; i++){
    it.next();
    it.remove();
}
```

# Collections Utilities

- **The `Collections` class contains `static` methods that operate on collections (similar to what the `Arrays` class does for arrays)**
  - Sort elements
  - Search for specific elements
  - Reverse, shuffle, or swap elements
  - Obtain thread-safe versions of a collection
  - Obtain "read-only" versions of a collection

- **Some methods work only with collections that implement specific interfaces**

-

# Collections Class Example

**Example**

```
// Create an ArrayList object
List<Integer> myList = new ArrayList<Integer>();

// Add elements to the list
myList.add(23);
myList.add(45);
myList.add(149);
myList.add(25);

// Element order is [23, 45, 149, 25]

// Randomly shuffle the list elements
Collections.shuffle(myList);

// Element order is now [149, 25, 23, 45]

// Reverse the order of the list elements
Collections.reverse(myList);

// Element order is now [45, 23, 25, 149]

// Swap the element at index 1 with the element at
// index 3
Collections.swap(myList,1,3);

// Element order is now [45, 149, 25, 23]

// Make the list unmodifiable
myList = Collections.unmodifiableList(myList);

// Runtime error if add() or remove() methods are
// called using the myList reference
```