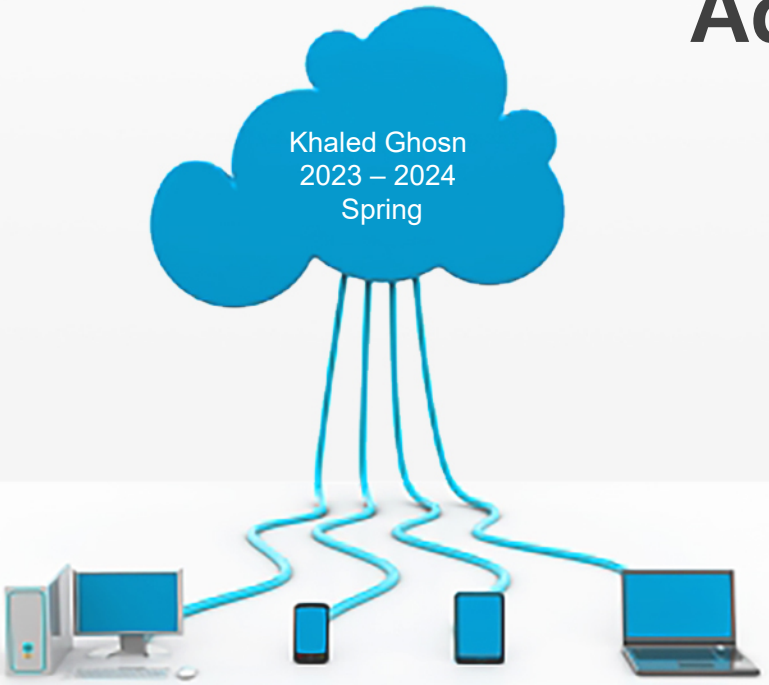
ARTS, SCIENCES & TECHNOLOGY UNIVERSITY IN LEBANON

AUL

CSC320 & CCE417

Advanced Programming using Java



Khaled Ghosn
2023 – 2024
Spring

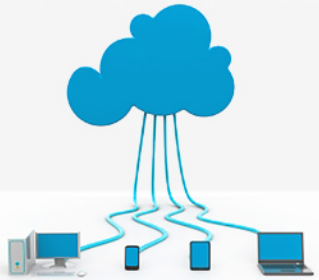
Introduction

Course Description

Welcome to the "**Advanced Programming** using java" course.

Based on OOP (P II course), this course presents how to write a more interactive programs, with the following features:

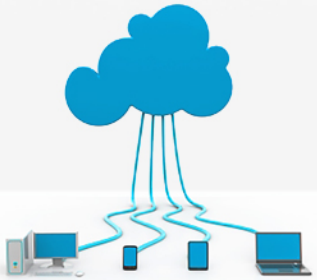
- Rich with graphical user interfaces through which users can interact with the system
- Able to handle exceptions for all possible errors
- Stores and retrieves data



Course Description

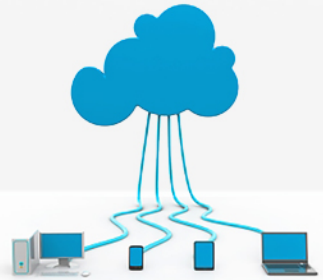
Goals:

- Identify GUI components and indicate their use cases
- Describe events associated by several types of classes
- Create desktop applications with rich and interactive graphical user interfaces
- List, differentiate, and construct several types of exceptions
- Describe and produce text and binary files to store and retrieve data
- Identify and describe the differences between serial access and random access files and their usages in various applications
- Create connection between java program and different DBMSs and exploit the connection to retrieve and store data efficiently

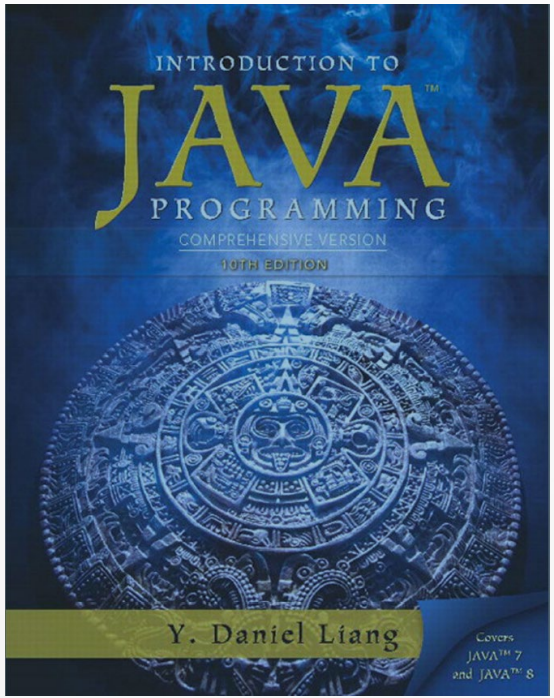


Course Outline

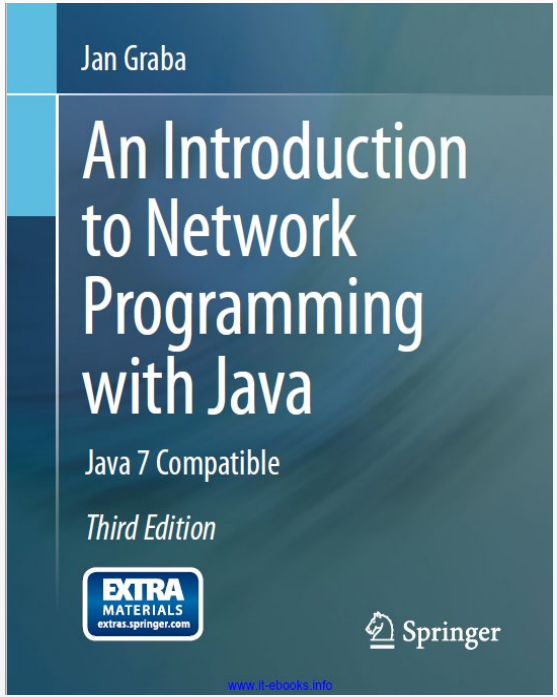
1. Course overview
2. GUI using JavaFX, Swing, and AWT packages
3. JavaFX components
4. JavaFX handling events
5. JavaFX animations
6. Exception handling
7. Files (Serial, Random Access)
8. Java Database Connectivity



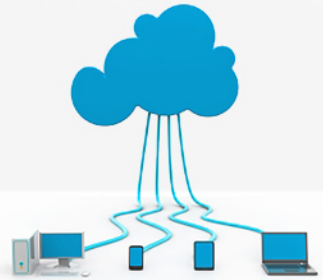
Textbook



INTRODUCTION TO JAVA PROGRAMMING
COMPREHENSIVE VERSION
Tenth Edition

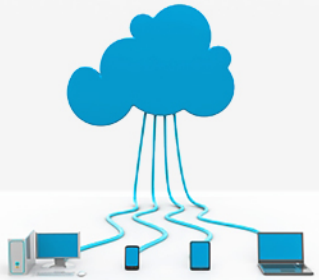


An Introduction to Network Programming with Java
Java 7 Compatible
Third Edition



Grade Distribution

Attendance	5 %
Week 3 Evaluation	3 %
Week 5 Evaluation	5 %
Week 7 Evaluation	6 %
Midterm	20%
Week 10 Evaluation	7 %
Week 12 Evaluation	7 %
Week 14 Evaluation	7 %
Final Exam	40 %



JAVA

You will be using the **Java** programming language in this course

This course does not teach Java programming

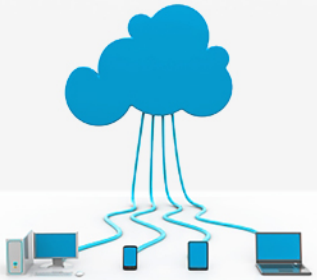
- You will use Java to demonstrate your knowledge in this course

One lecture covers

- Revision about Classes using Java

Commenting code is necessary for engineers:

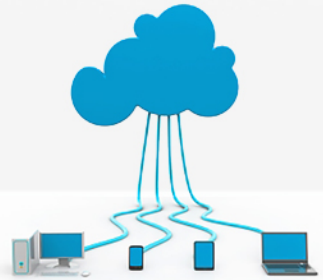
- Engineers who do not comment code will not encourage employees and contracted programmers to comment their code
- This will lead to significant additional costs



JAVA

WHY WE LOVE IT

- Versatility (e.g. able to adapt)
- Object-oriented programming
  (paradigm based on the concept of "objects")
- Great place to start
- Excellent online documentation
- Easy to teach & easy to learn
- Widely used



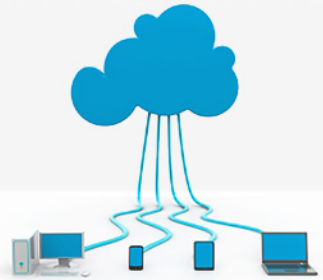
JAVA

Make sure you already have all the needed development tools:

- Java Development Kit (JDK) for Java Standard Edition (SE)
- An Integrated Development Environment (IDE) such as NetBeans or Eclipse
- A java-fx plugin is integrated with your IDE

Get familiar with the IDE basic tasks:

- project creation
- basic file and package manipulation
- code editing and completion
- most used keyboard shortcuts
- running and debugging your code.



JAVA

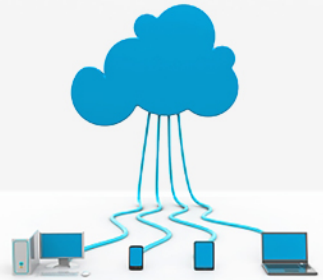
Revision

basic java principles

variables, conditions, loops (iterations), arrays, functions ...

OOP:

- classes vs objects
- constructors
- encapsulation
- overloading
- inheritance
- overriding
- interfaces
- abstract classes
- polymorphism



JAVA

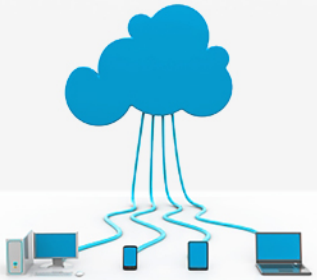
ENVIRONMENT SETUP - JDK

- Download Java Development Kit 8 for your corresponding OS.

https://www.oracle.com/java/technologies/javase/javase-jdk8-downloads.html

- JDK includes a runtime environment but JRE doesn't.

- To verify that java is correctly installed, go to Start > cmd.exe and run "java –version"



JAVA

ENVIRONMENT SETUP - JDK

NETBEANS

1. Download the latest version of NetBeans.
2. In order to get the IDE up and running with JavaFX, please follow step-by-step what is mentioned in the corresponding YouTube video.
3. In NetBeans, create a New Project, select "Java With Ant", then "JavaFX", and choose JavaFX Application. Rename your project and press Finish.

https://netbeans.org/

https://www.youtube.com/watch?v=UlobR93nJow

ECLIPSE

1. Download the latest version of Eclipse IDE.
2. In order to get the IDE up and running with JavaFX, please follow step-by-step what is mentioned in the corresponding YouTube video.
3. In Eclipse, create a New Project, select JavaFX project, rename it and press Finish.

https://www.eclipse.org/downloads/packages/installer

https://www.eclipse.org/efxclipse/install.html

https://gluonhq.com/products/javafx/

https://www.youtube.com/watch?v=bC4XB6JAaoU

Rules

Improving Your Performance

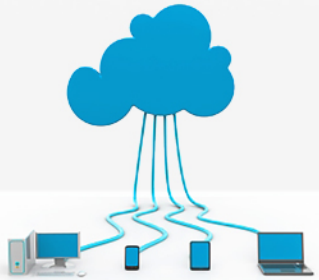
The human brain can retain approximately 5-9 independent items of information in its short-term memory

> George Miller, *The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information*, Psychological Review, Vol.63 pp.81–97, 1956

The introduction of new information causes the brain to discard an item currently in your short-term memory

- For example, consider the 12 words which will appear on the next sequence of screens

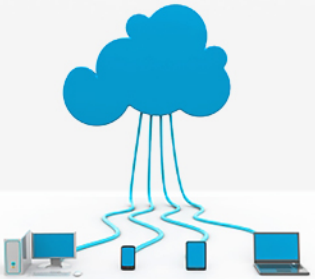
To transfer information from your short-term memory to your long- term memory, that information must be imposed on your mind at least three times



Rules

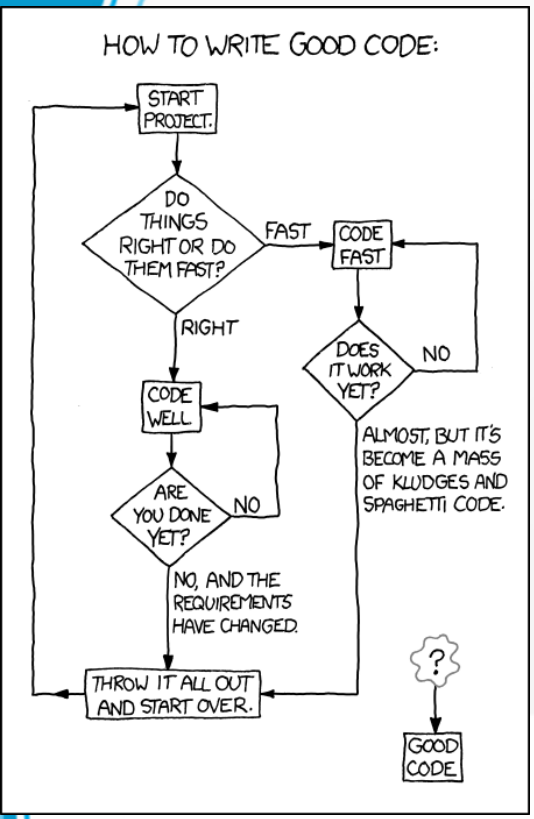
WORKLOAD & RULES

- Course consists of 42 hours, i.e. 14 weeks - 28 (1.5 hr.) sessions
- Each session requires 2 to 6 hours of work:
- Revising concepts covered in class
- Practicing exercises solved in class
- Homework solving (if any)
- Your presence in class is extremely important, not just for attendance
- Assignments should be solved individually; plagiarism will not be tolerate, a deduction of grades will be applied for BOTH parties involved
- Send formal emails! Otherwise, will be ignored!
  - put in the e-mail subject: [AP]
  - example: [AP] homework1



Rules

Improving Your Performance



You should always try the following:

- Look at the slides before class
- Attend lectures
  - You see the information again with commentary
- Review the lecture during the evening
  - Rewrite and summarize the slides in **your** words

In addition to this, you should:

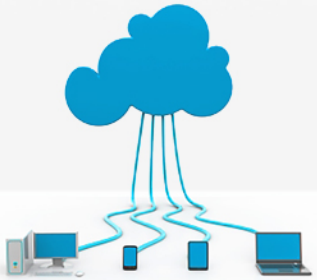
- Get a reasonable nights sleep (apparently this is when information is transferred to your long-term memory), and
- Eat a good breakfast (also apparently good for the memory)

Rules

Plagiarism

All assignments must be done individually:

- You may not copy code directly from any other source
- Plagiarism detection software will be used on all of the assignments
- If you viewed another code (from books or lecture notes), you must include a reference in your assignments
- You may not share code with any other students by transmitting completed functions to your peers
- You may discuss assignments together and help another student debug his or her code; however, you cannot dictate or give the exact solution
- When one student copies from another student, both students are responsible (exceptions are made for outright theft)



Rules

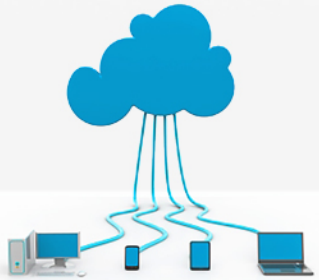
Plagiarism

All assignments must be done individually:

- The penalty for plagiarism on an assignment is a mark of 0
- Regardless if Projects are counted or not
- A student who cheats must receive a grade lower than a student who did not hand in a project

The best way to avoid plagiarism is:

- review the course Programming 2
- read the assignments as soon as they are available
- start the assignment so that there is sufficient time to contact me if you have difficulty
- do not give your code to anyone



Instructor Contact

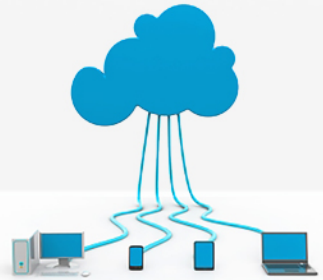
Assignments can be sent by e-mails via **Google Classroom**; otherwise will be ignored!

Facebook & **WhatsApp** are never accepted for sending assignments, otherwise you are welcome to contact for any assistant.

Do not hesitate to contact me for any assistance through the e-mail:

KhaledGhosn@Hotmail.com

kg002@live.aul.edu.lb

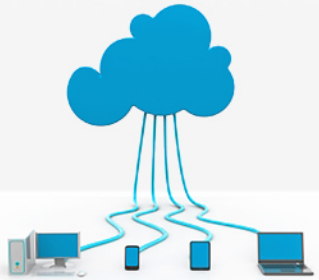


Usage Notes

These slides are made publicly available for anyone to use

If you choose to use them, or a part thereof, I ask only three things:

- that you inform me that you are using the slides,
- that you acknowledge my work, and
- that you alert me of any mistakes which I made or changes which you make, and allow me the option of incorporating such changes (with an acknowledgment) in my set of slides



Sincerely,
Ghosn Khaled
KhaledGhosn@Hotmail.com

Revision on Basic Java Programming

Loops

Motivation:

Suppose that you need to print a string (e.g., "Welcome to Java!") a hundred times. It would be tedious to have to write the following statement a hundred times:

```
System.out.println("Welcome to Java!");
```

So, how do you solve this problem?

Opening Problem

Problem:

100 times

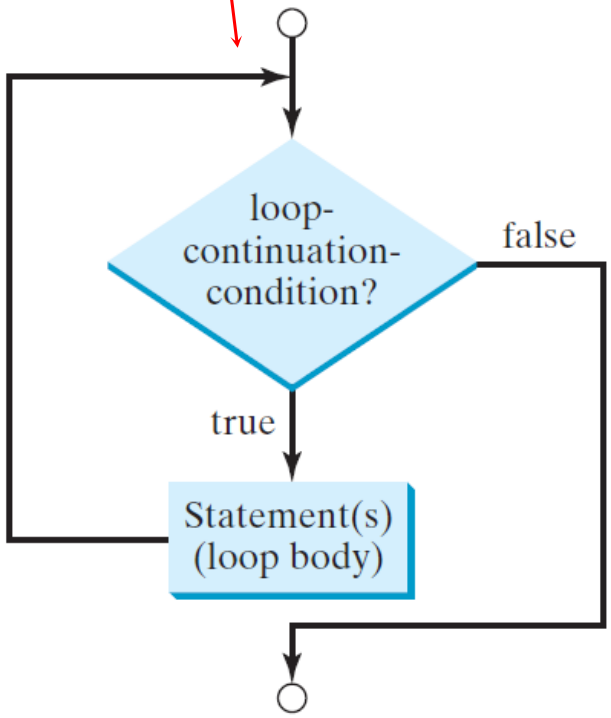
```
System.out.println("Welcome to Java!");
System.out.println("Welcome to Java!");
System.out.println("Welcome to Java!");
System.out.println("Welcome to Java!");
System.out.println("Welcome to Java!");
System.out.println("Welcome to Java!");

...

...

...
System.out.println("Welcome to Java!");
System.out.println("Welcome to Java!");
System.out.println("Welcome to Java!");
```

Introducing while Loops

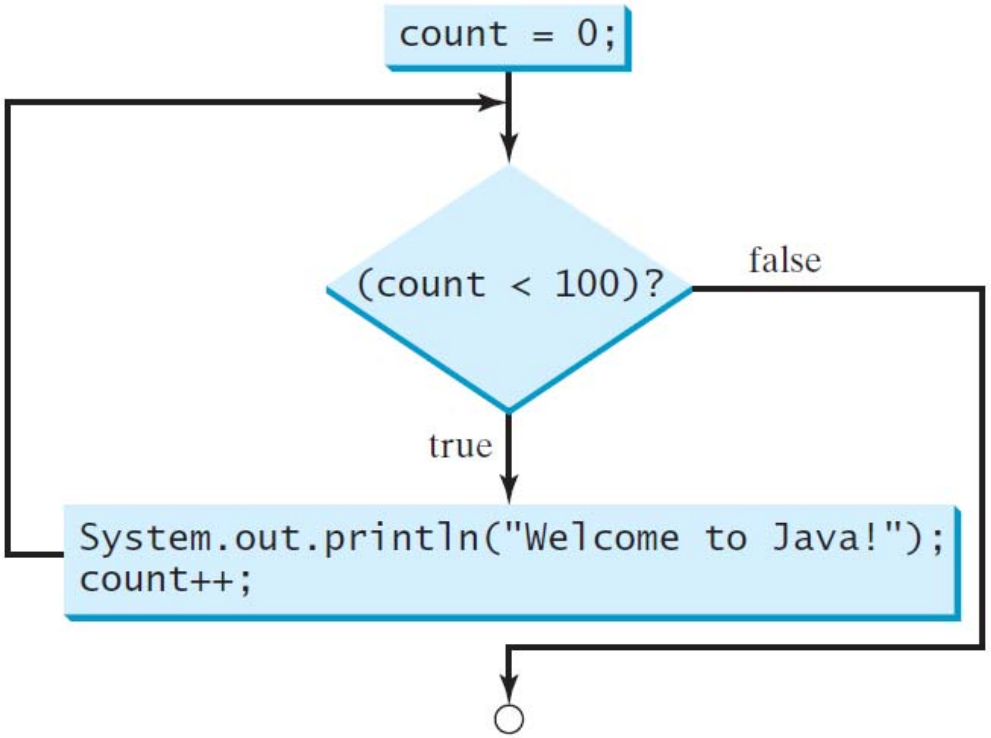
```
int count = 0;
while (count < 100) {
  System.out.println("Welcome to Java");
  count++;
}
```

`while` Loop Flow Chart

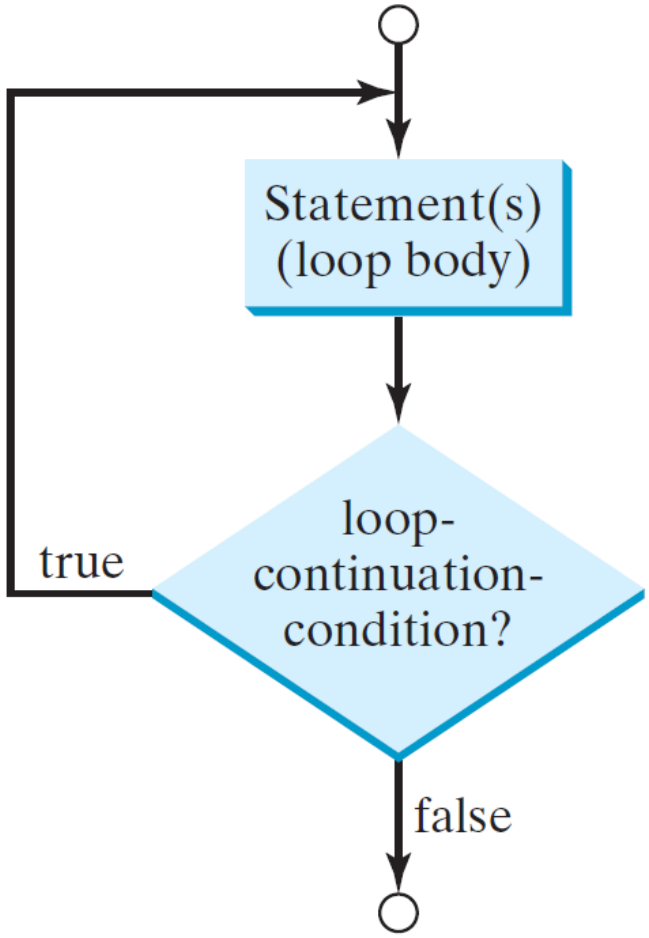
```
while (loop-continuation-condition) {
  // loop-body;
  Statement(s);
}
```




```
int count = 0;
while (count < 100) {
  System.out.println("Welcome to Java!");
  count++;
}
```



do-while Loop

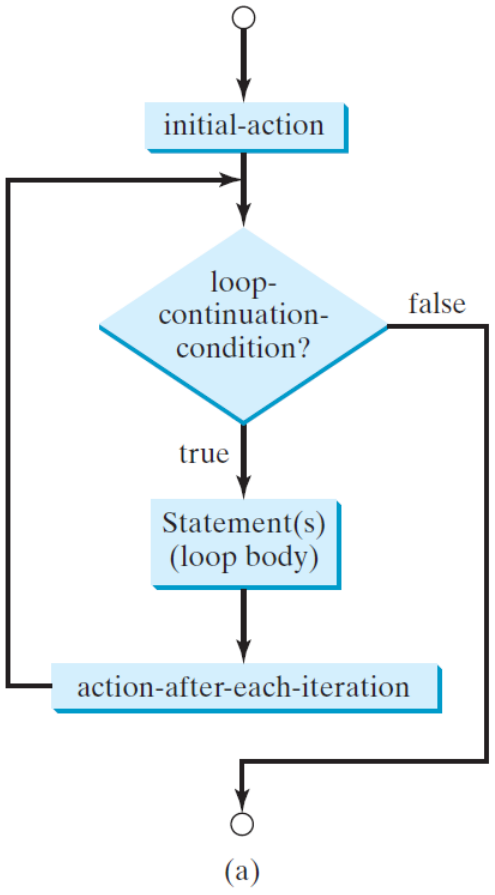



```
do {
  // Loop body;
  Statement(s);
} while (loop-continuation-condition);
```

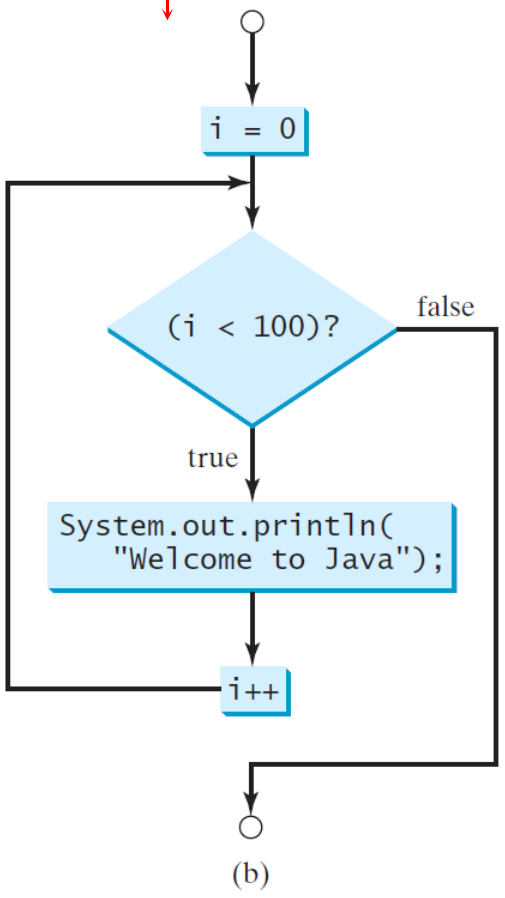
for Loops

```
for (initial-action; loop-
    continuation-condition; action-
    after-each-iteration) {
  // loop body;
  Statement(s);
}
```

```
int i;
for (i = 0; i < 100; i++) {
  System.out.println(
    "Welcome to Java!");
}
```



(a)



(b)

Note

The initial-action in a for loop can be a list of zero or more comma-separated expressions. The action-after-each-iteration in a for loop can be a list of zero or more comma-separated statements. Therefore, the following two for loops are correct. They are rarely used in practice, however.

```
for (int i = 1; i < 100; System.out.println(i++));
```

```
for (int i = 0, j = 0; (i + j < 10); i++, j++) {
  // Do something
}
```

Note

If the loop-continuation-condition in a for loop is omitted, it is implicitly true. Thus the statement given below in (a), which is an infinite loop, is correct. Nevertheless, it is better to use the equivalent loop in (b) to avoid confusion:

```
for ( ; ; ) {
  // Do something
}
```

(a)

Equivalent

```
while (true) {
  // Do something
}
```

(b)

break

```
public class TestBreak {
  public static void main(String[] args) {
    int sum = 0;
    int number = 0;

    while (number < 20) {
      number++;
      sum += number;
      if (sum >= 100)
        break;
    }

    System.out.println("The number is " + number);
    System.out.println("The sum is " + sum);
  }
}
```

continue

```
public class TestContinue {
  public static void main(String[] args) {
    int sum = 0;
    int number = 0;

    while (number < 20) {
      number++;
      if (number == 10 || number == 11)
        continue;
      sum += number;
    }

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

Methods

Defining Methods

A method is a collection of statements that are grouped together to perform an operation.

Define a method

```
public static int max(int num1, int num2) {

  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

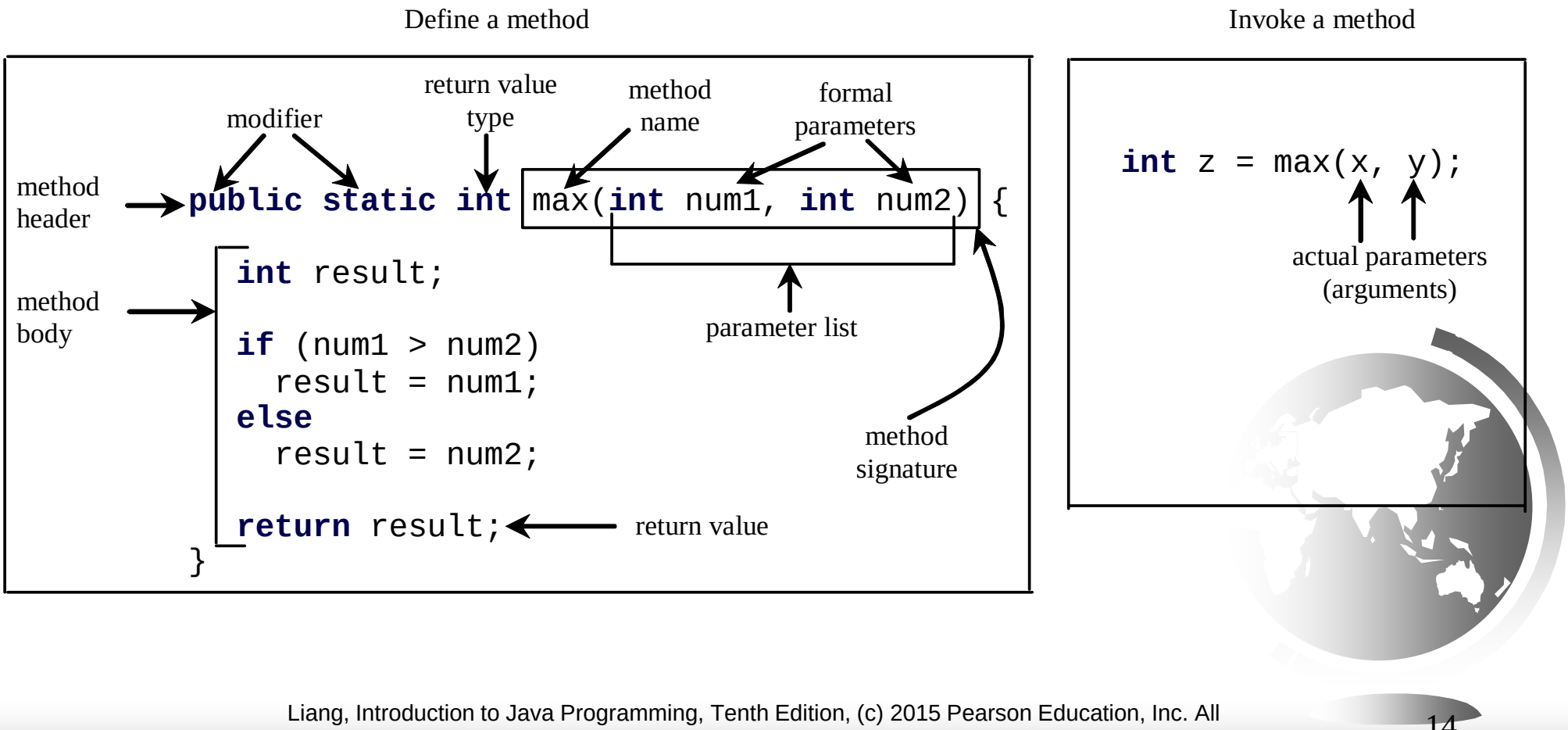
Invoke a method

```
int z = max(x, y);
```

actual parameters (arguments)

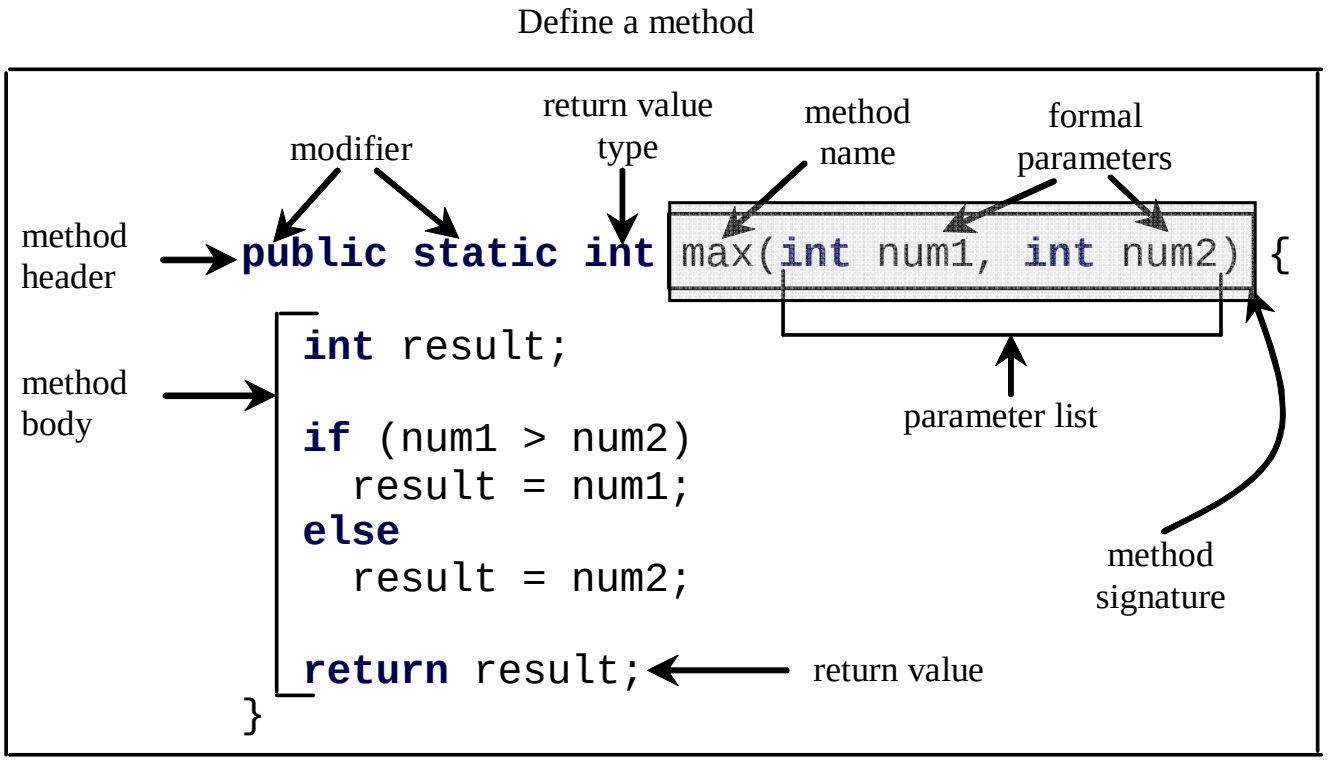
Defining Methods

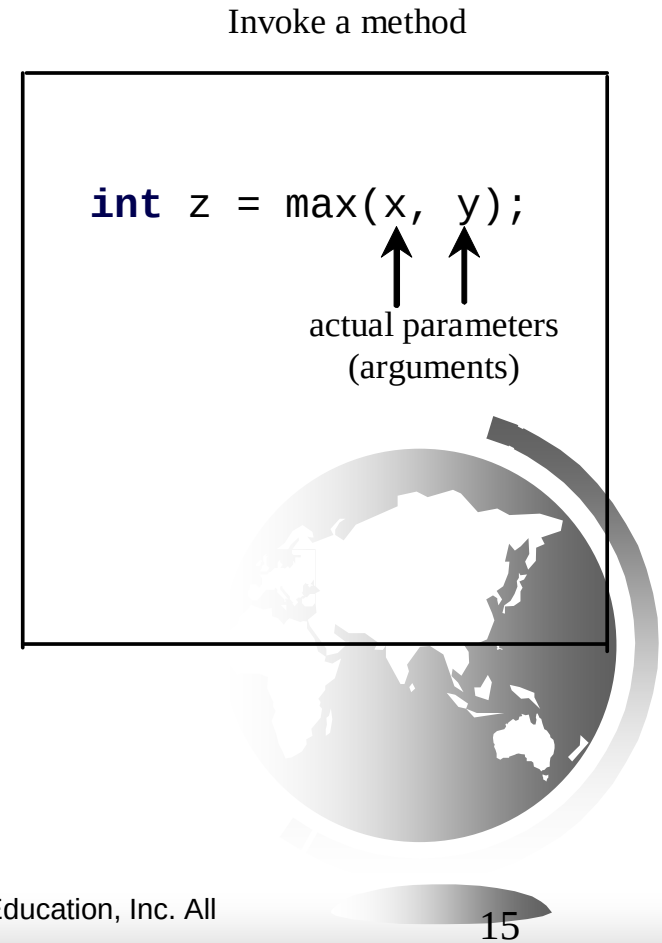
A method is a collection of statements that are grouped together to perform an operation.



Method Signature

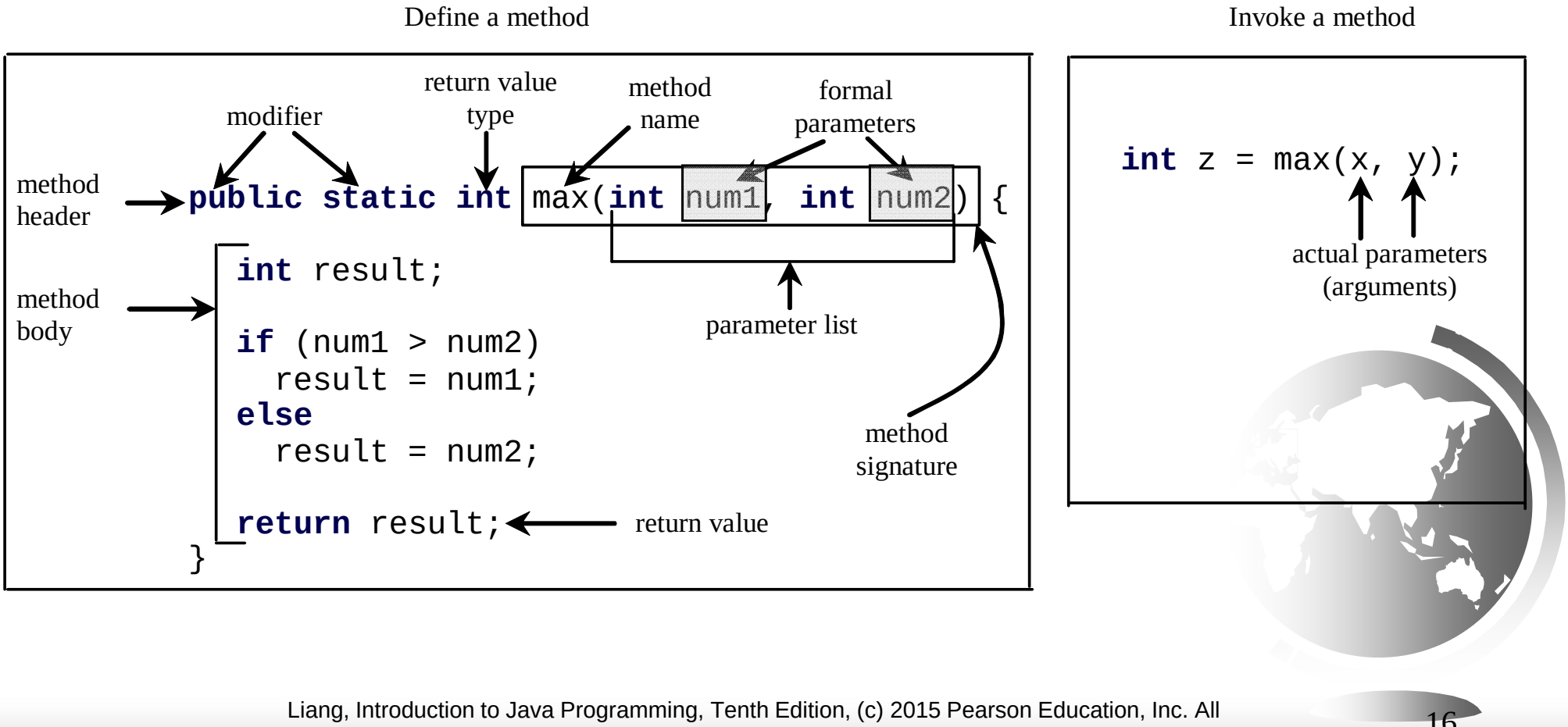
Method signature is the combination of the method name and the parameter list.





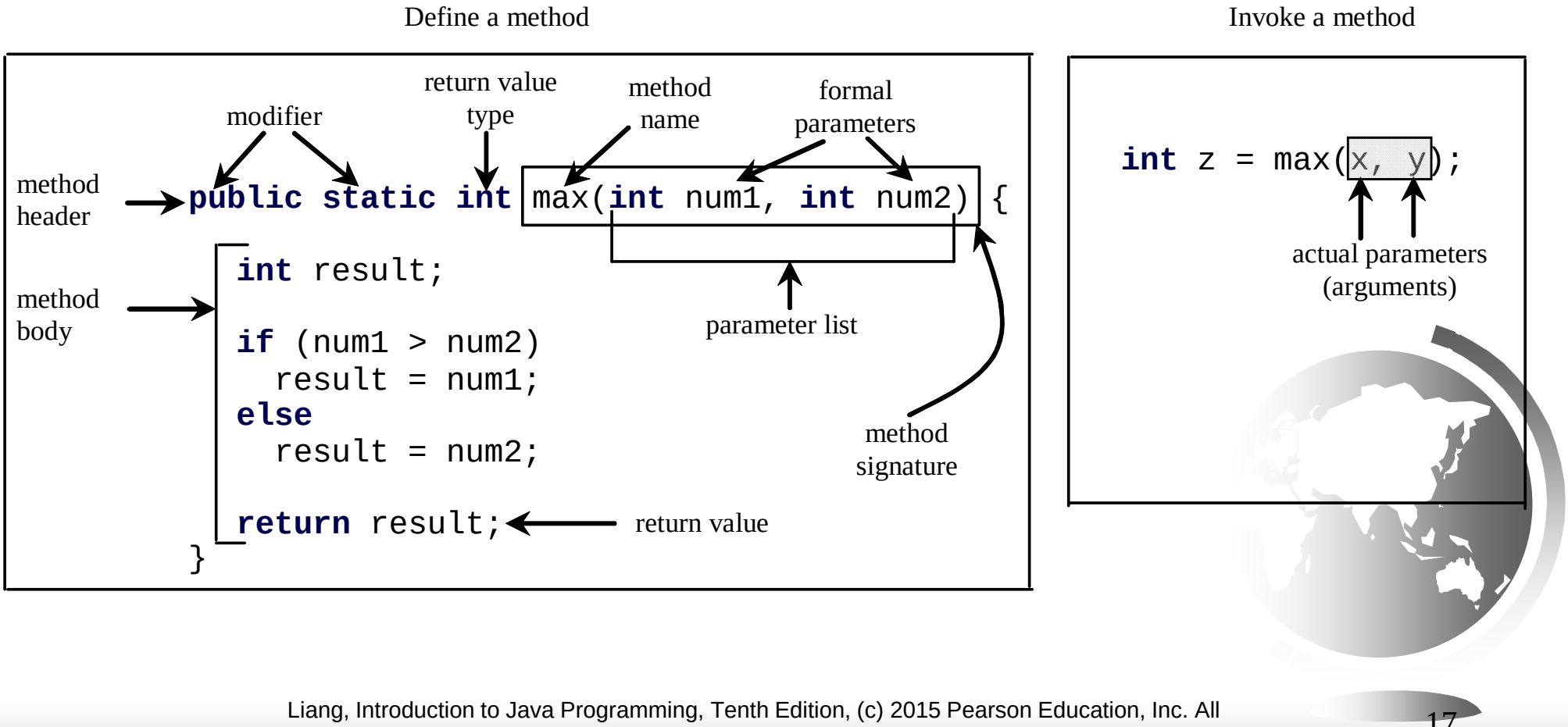
Formal Parameters

The variables defined in the method header are known as *formal parameters*.



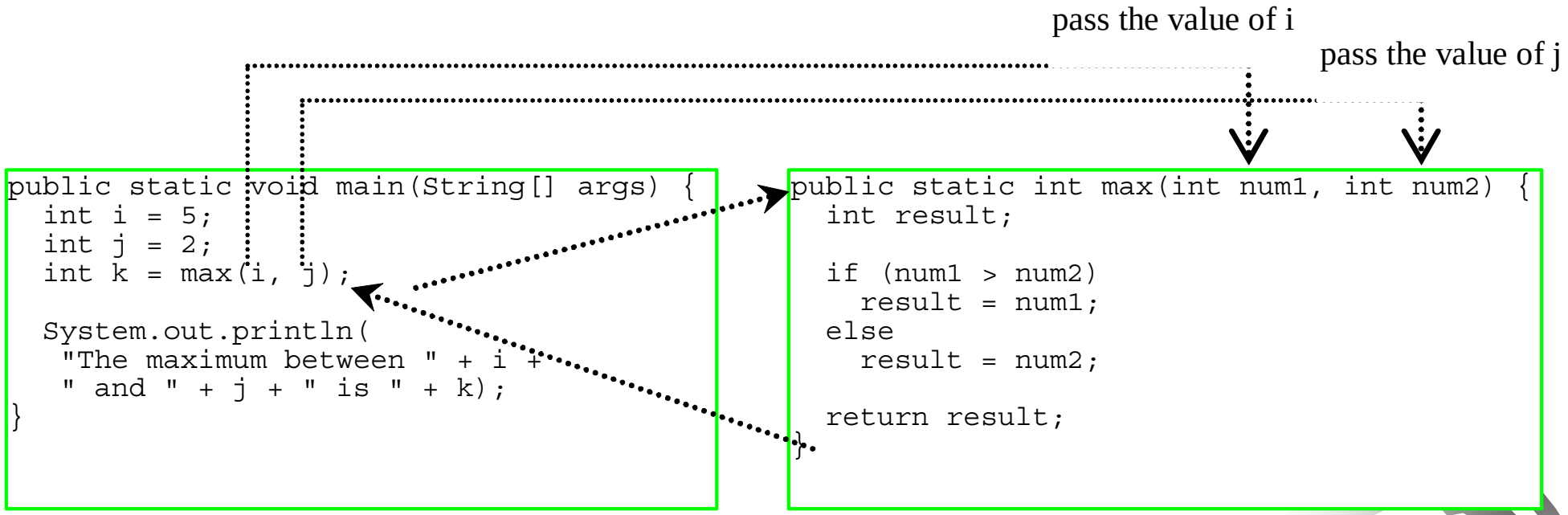
Actual Parameters

When a method is invoked, you pass a value to the parameter. This value is referred to as *actual parameter or argument*.



animation

Calling Methods



CAUTION

A return statement is required for a value-returning method. The method shown below in (a) is logically correct, but it has a compilation error because the Java compiler thinks it possible that this method does not return any value.

```
public static int sign(int n) {
  if (n > 0)
    return 1;
  else if (n == 0)
    return 0;
  else if (n < 0)
    return -1;
}
```

(a)

Should be

```
public static int sign(int n) {
  if (n > 0)
    return 1;
  else if (n == 0)
    return 0;
  else
    return -1;
}
```

(b)

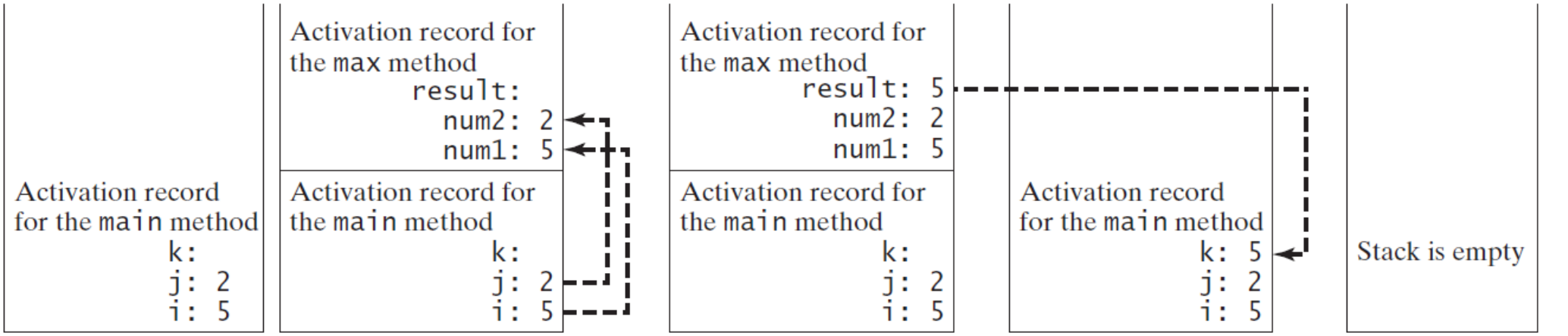
To fix this problem, delete *if (n < 0)* in (a), so that the compiler will see a return statement to be reached regardless of how the if statement is evaluated.

Call Stacks

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
    "The maximum between " + i +
    " and " + j + " is " + k);
}
```

```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

Call Stacks



(a) The `main` method is invoked.

(b) The `max` method is invoked.

(c) The `max` method is being executed.

(d) The `max` method is finished and the return value is sent to `k`.

(e) The `main` method is finished.

Overloading Methods

Overloading the `max` Method

```
public static double max(double num1, double
  num2) {
   if (num1 > num2)
     return num1;
   else
     return num2;
}
```

TestMethodOverloading Run

Scope of Local Variables

A local variable: a variable defined inside a method.

Scope: the part of the program where the variable can be referenced.

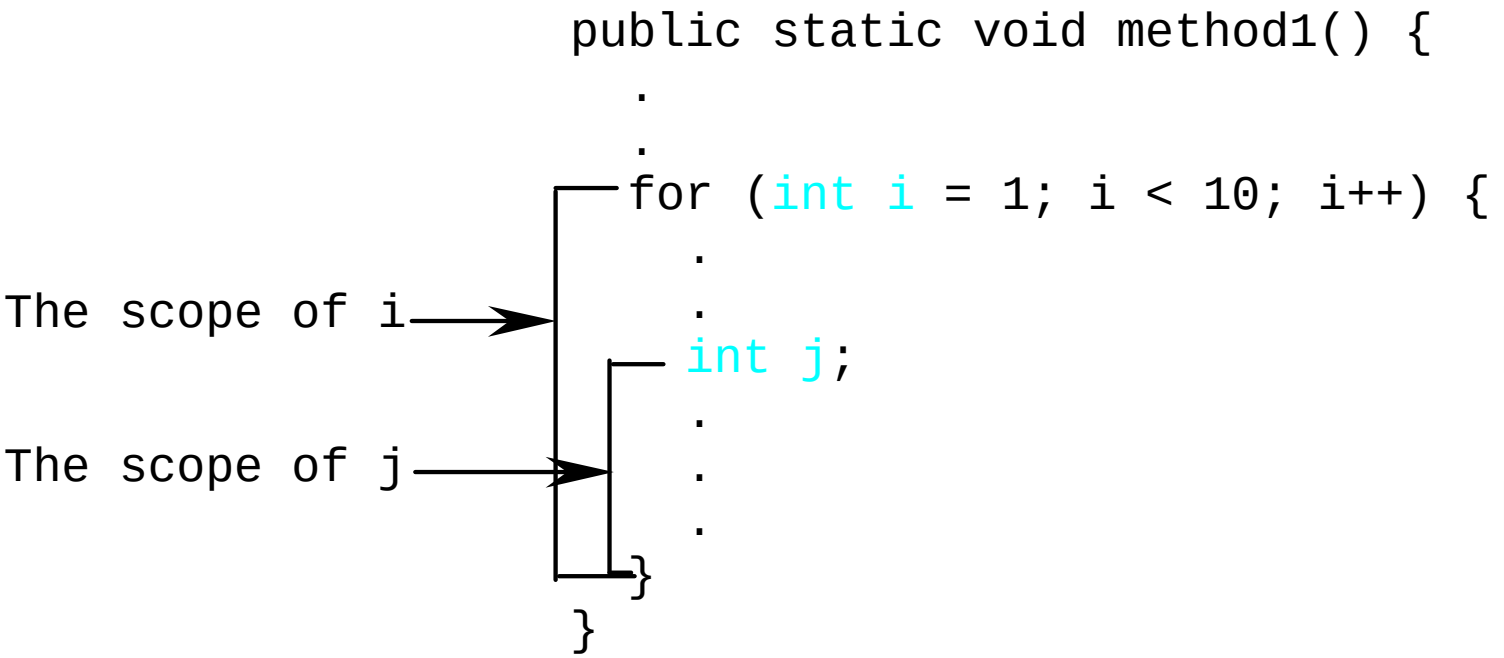
The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable. A local variable must be declared before it can be used.

Scope of Local Variables, cont.

You can declare a local variable with the same name multiple times in different non-nesting blocks in a method, but you cannot declare a local variable twice in nested blocks.

Scope of Local Variables, cont.

A variable declared in the initial action part of a for loop header has its scope in the entire loop. But a variable declared inside a for loop body has its scope limited in the loop body from its declaration and to the end of the block that contains the variable.



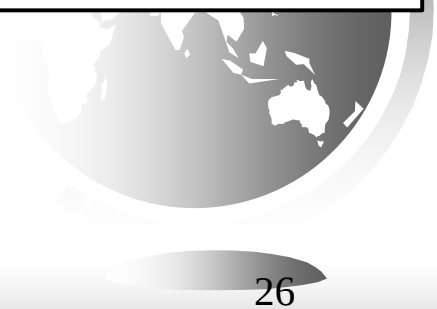
Scope of Local Variables, cont.

It is fine to declare i in two non-nesting blocks

```
public static void method1() {
  int x = 1;
  int y = 1;

  for (int i = 1; i < 10; i++) {
    x += i;
  }

  for (int i = 1; i < 10; i++) {
    y += i;
  }
}
```

It is wrong to declare i in two nesting blocks

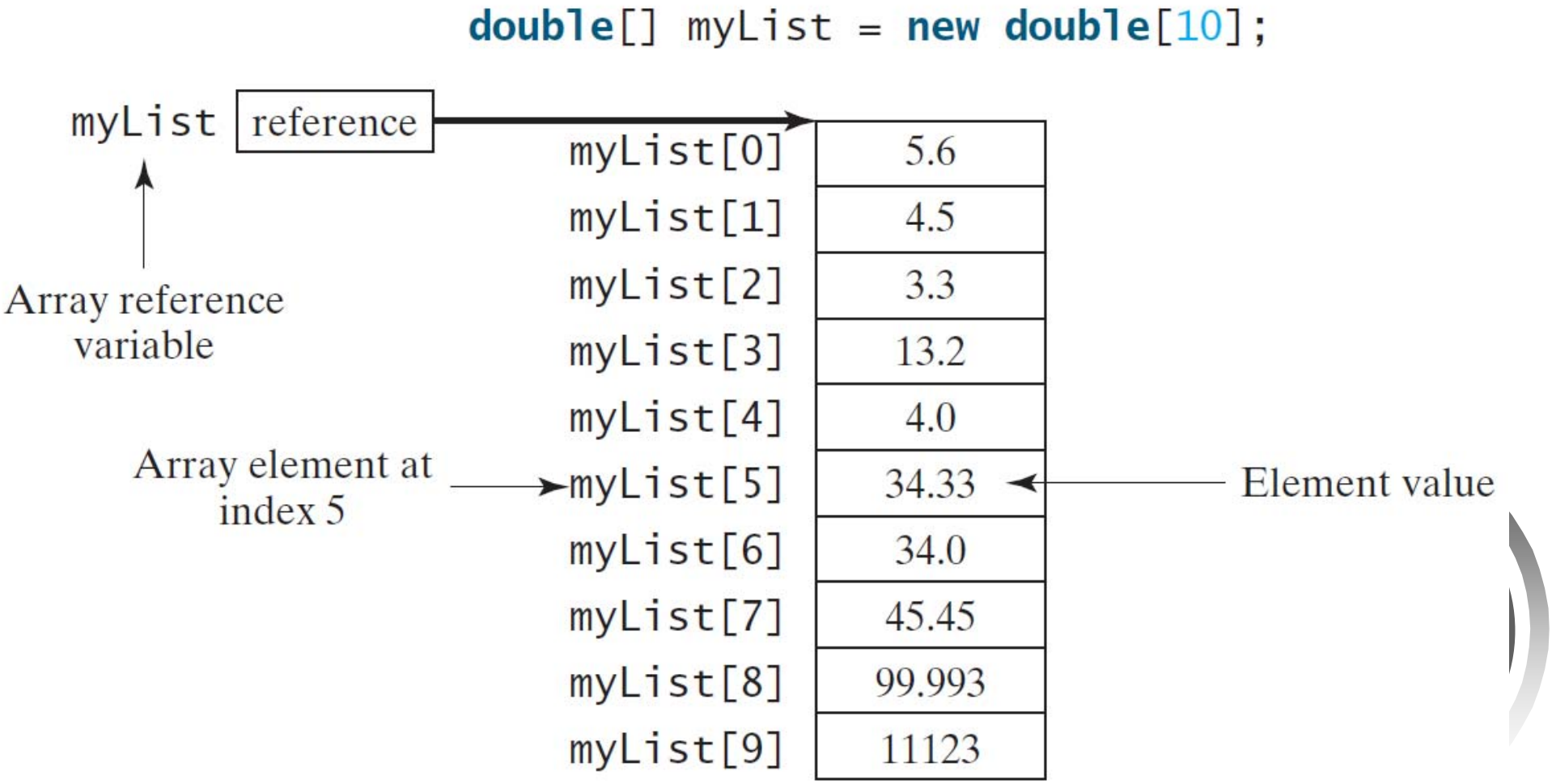
```
public static void method2() {

  int i = 1;
  int sum = 0;

  for (int i = 1; i < 10; i++)
    sum += i;
  }

}
```



Arrays

Introducing Arrays

Array is a data structure that represents a collection of the same types of data.



Declaring Array Variables

- `datatype[] arrayRefVar;`

  Example:

  ```
  double[] myList;
  myList = new double[10];
  ```

- `datatype arrayRefVar[];` // This style is allowed, but not preferred

  Example:

  ```
  double myList[];
  ```

Declaring and Creating in One Step

- ```
  datatype[] arrayRefVar = new
     datatype[arraySize];
  ```

  ```
  double[] myList = new double[10];
  ```

- ```
  datatype arrayRefVar[] = new
    datatype[arraySize];
  ```

  ```
  double myList[] = new double[10];
  ```

The Length of an Array

Once an array is created, its size is fixed. It cannot be changed. You can find its size using

```
arrayRefVar.length
```

For example,

```
myList.length returns 10
```

Indexed Variables

The array elements are accessed through the index. The array indices are *0-based*, i.e., it starts from 0 to arrayRefVar.length-1. In the example in Figure 6.1, myList holds ten double values and the indices are from 0 to 9.

Each element in the array is represented using the following syntax, known as an *indexed variable*:

```
arrayRefVar[index];
```

Initializing arrays with input values

```
java.util.Scanner input = new java.util.Scanner(System.in);
System.out.print("Enter " + myList.length + " values: ");
for (int i = 0; i < myList.length; i++)
  myList[i] = input.nextDouble();
```

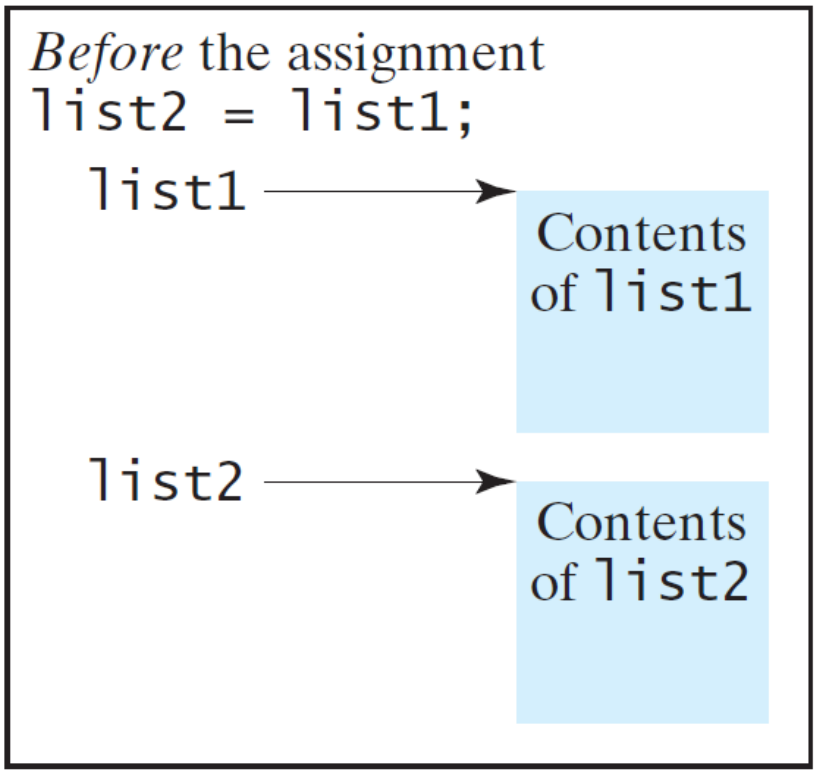
Initializing arrays with random values

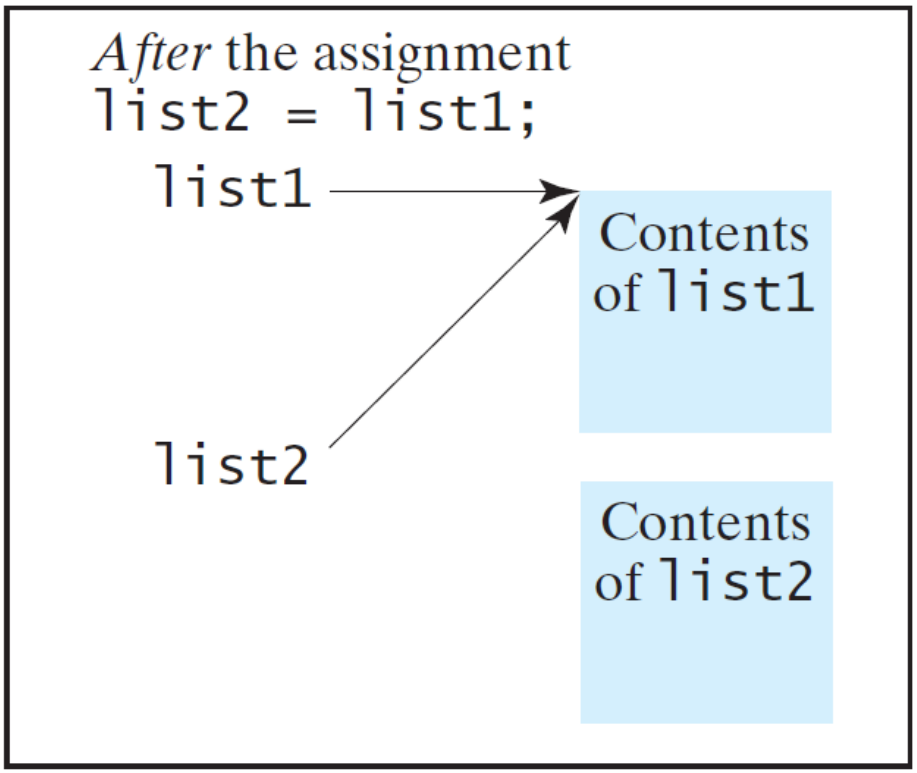
```
for (int i = 0; i < myList.length; i++) {
  myList[i] = Math.random() * 100;
}
```

Copying Arrays

Often, in a program, you need to duplicate an array or a part of an array. In such cases you could attempt to use the assignment statement (=), as follows:

list2 = list1;





Copying Arrays

Using a loop:

```
int[] sourceArray = {2, 3, 1, 5, 10};
int[] targetArray = new
  int[sourceArray.length];

for (int i = 0; i < sourceArrays.length; i++)
   targetArray[i] = sourceArray[i];
```

Classes

Classes are constructs that define objects of the same type. A Java class uses variables to define data fields and methods to define behaviors. Additionally, a class provides a special type of methods, known as constructors, which are invoked to construct objects from the class.

Classes

```
class Circle {
  /** The radius of this circle */
  double radius = 1.0;

  /** Construct a circle object */
  Circle() {
  }

  /** Construct a circle object */
  Circle(double newRadius) {
    radius = newRadius;
  }

  /** Return the area of this circle */
  double getArea() {
    return radius * radius * 3.14159;
  }
}
```

Data field

Constructors

Method

Constructors

```
Circle() {
}
```

Constructors are a special kind of methods that are invoked to construct objects.

```
Circle(double newRadius) {
  radius = newRadius;
}
```

Constructors do not have a return type—not even void.

Constructors are invoked using the new operator when an object is created. Constructors play the role of initializing objects.

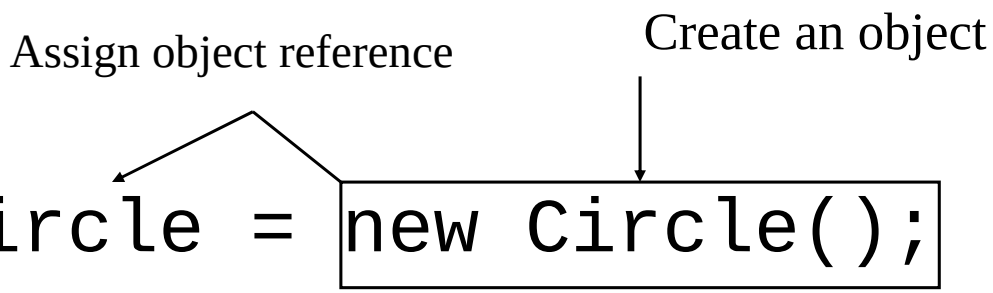
Default Constructor

A class may be defined without constructors. In this case, a no-arg constructor with an empty body is implicitly defined in the class. This constructor, called *a default constructor*, is provided automatically *only if no constructors are explicitly defined in the class*.

Declaring/Creating Objects in a Single Step

```
ClassName objectRefVar = new ClassName();
```

Example:



Accessing Object's Members

- Referencing the object's data:

  `objectRefVar.data`

  e.g., `myCircle.radius`

- Invoking the object's method:

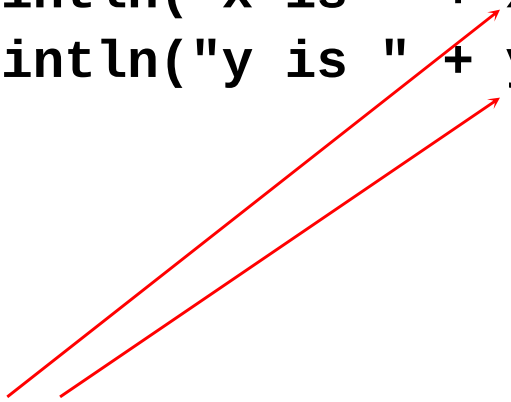
  `objectRefVar.methodName(arguments)`

  e.g., `myCircle.getArea()`

Caution

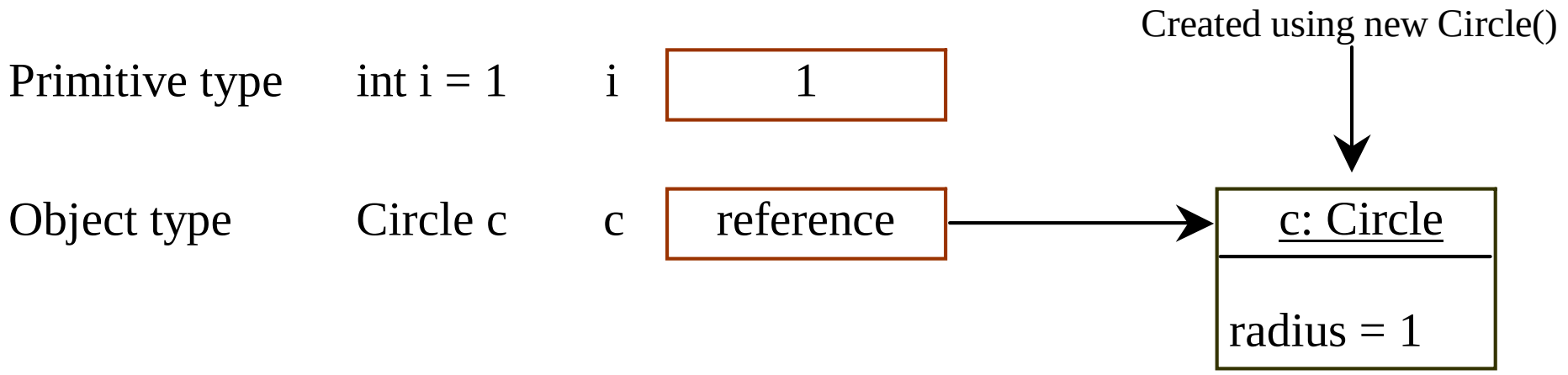
Java assigns no default value to a local variable inside a method.

```
public class Test {
  public static void main(String[] args) {
    int x; // x has no default value
    String y; // y has no default value
    System.out.println("x is " + x);
    System.out.println("y is " + y);
  }
}
```



Compile error: variable not initialized

Differences between Variables of Primitive Data Types and Object Types



Static Variables, Constants, and Methods

Static variables are shared by all the instances of the class.

Static methods are not tied to a specific object.

Static constants are final variables shared by all the instances of the class.

Visibility Modifiers and Accessor/Mutator Methods

By default, the class, variable, or method can be accessed by any class in the same package.

- `public`

  The class, data, or method is visible to any class in any package.

- `private`

  The data or methods can be accessed only by the declaring class.

The get and set methods are used to read and modify private properties.

Class Relationships

Composition

Inheritance (Chapter 13)

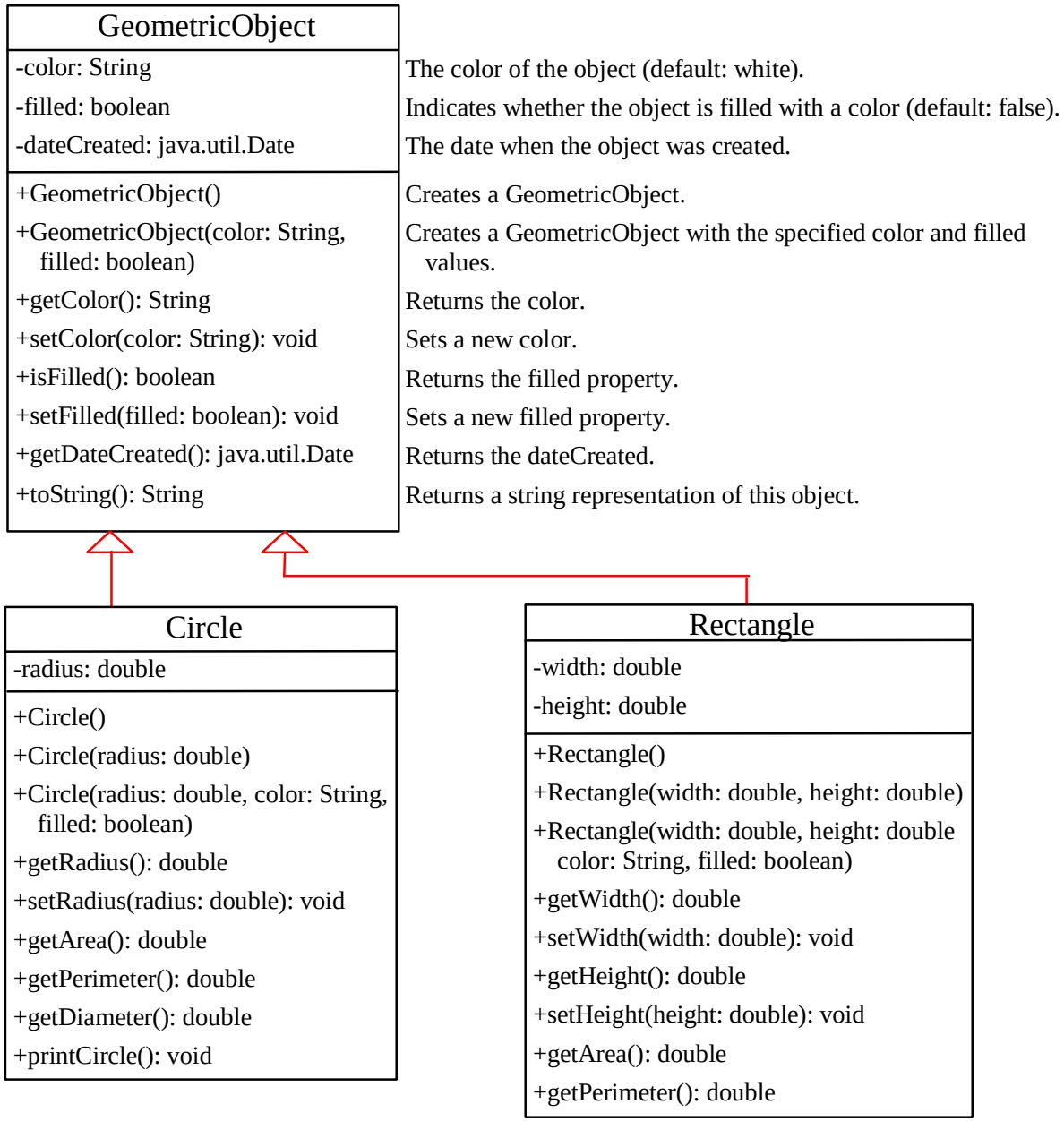
Composition represents an ownership relationship between two objects. The Child (owned) class is usually represented as a data field in the Parent (owner) class.

```
public class Name {
  ...
}
```

```
public class Student {
  private Name name;
  private Address address;

  ...
}
```

```
public class Address {
  ...
}
```

Inheritance: Superclasses and Subclasses



Defining a Subclass

A subclass inherits from a superclass. You can also:

- Add new properties
- Add new methods
- Override the methods of the superclass (you must override each abstract method)

Are superclass's Constructor Inherited?

No. They are not inherited.

They are invoked explicitly or implicitly.

Explicitly using the super keyword.

A constructor is used to construct an instance of a class. Unlike properties and methods, a superclass's constructors are not inherited in the subclass. They can only be invoked from the subclasses' constructors, using the keyword super. *If the keyword super is not explicitly used, the superclass's no-arg constructor is automatically invoked.*

Superclass's Constructor Is Always Invoked

A constructor may invoke an overloaded constructor or its superclass's constructor. If none of them is invoked explicitly, the compiler puts super() as the first statement in the constructor. For example,

```
public A() {
}
```

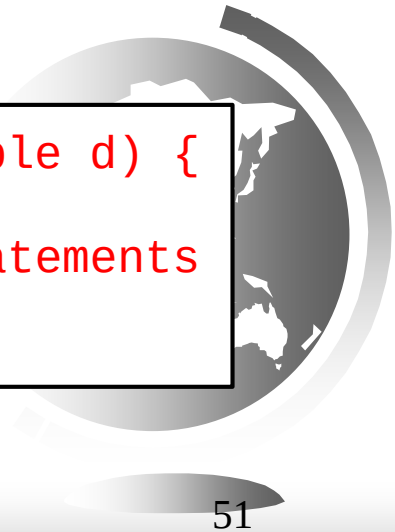
is equivalent to

```
public A() {
  super();
}
```

```
public A(double d) {
  // some statements
}
```

is equivalent to

```
public A(double d) {
  super();
  // some statements
}
```



Using the Keyword `super`

The keyword `super` refers to the superclass of the class in which `super` appears. This keyword can be used in two ways:

- To call a superclass constructor
- To call a superclass method

Overriding vs. Overloading

```java
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overrides the method in B
  public void p(double i) {
    System.out.println(i);
  }
}
```

```java
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overloads the method in B
  public void p(int i) {
    System.out.println(i);
  }
}
```

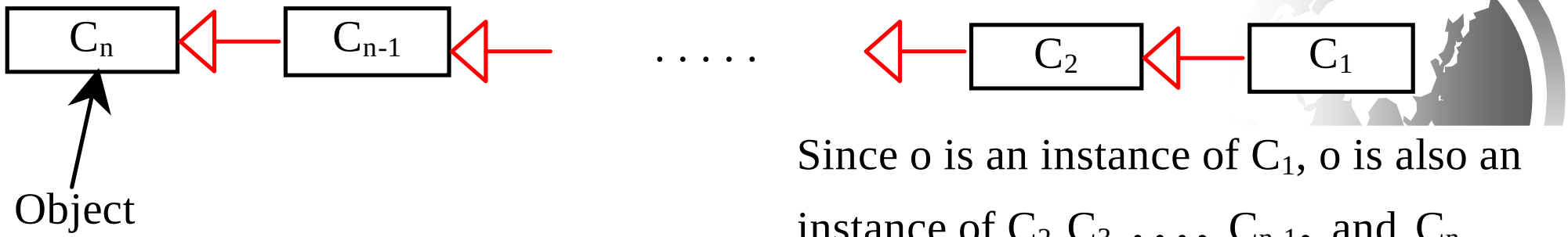
Polymorphism

```
class Person {
  public String toString() {
    return "Person";
  }
}

class Student extends Person {
  @Override
  public String toString() {
    return "Student";
  }
}


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

    //Polymorphism
    Person[] P = new Person[10];

    Person m = new Person();
    Student s = new Student();
    P[0] = m;
    P[1] = s;
    System.out.println(P[0].toString());
    System.out.println(P[1].toString());
  }
}
```

Polymorphism means that a variable of a supertype can refer to a subtype object. In other words, An object of a subtype can be used wherever its supertype value is required

Dynamic Binding

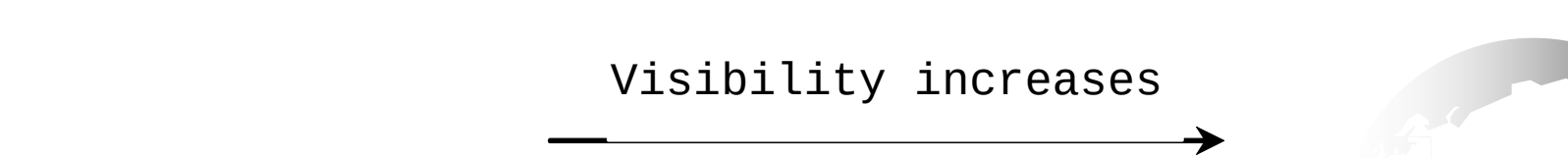
Dynamic binding works as follows: Suppose an object o is an instance of classes C_1, C_2, ..., C_{n-1}, and C_n, where C_1 is a subclass of C_2, C_2 is a subclass of C_3, ..., and C_{n-1} is a subclass of C_n. That is, C_n is the most general class, and C_1 is the most specific class. Suppose object o is of type C_1. If o invokes a method p, the JVM searches the implementation for the method p in C_1, C_2, ..., C_{n-1} and C_n, in this order, until it is found. Once an implementation is found, the search stops and the first-found implementation is invoked.



Since o is an instance of C_1, o is also an instance of C_2, C_3, . . . , C_{n-1}, and C_n

The `protected` Modifier

- The `protected` modifier can be applied on data and methods in a class. A protected data or a protected method in a public class can be accessed by any class in the same package or its subclasses, even if the subclasses are in a different package.
- private, default, protected, public



Accessibility Summary

Modifier on members in a class	Accessed from the same class	Accessed from the same package	Accessed from a subclass	Accessed from a different package
public	✓	✓	✓	✓
protected	✓	✓	✓	-
default	✓	✓	-	-
private	✓	-	-	-

The `final` Modifier

- The `final` class cannot be extended:

```
final class Math {
  ...
}
```

- The `final` variable is a constant:

```
final static double PI = 3.14159;
```

- The `final` method cannot be overridden by its subclasses.

JavaFX Basics

What is JavaFX

- JavaFX is a new framework for developing Java GUI programs.
- JavaFX is used for creating desktop applications, as well as rich internet applications.
- JavaFX supports Microsoft Windows, Linux, and macOS, and all web browsers that run on them.
- JavaFX applications could run on any desktop that has Java SE, on any browser that could run Java EE, or on any mobile phone that could run Java ME.

What is JavaFX

JavaFX vs Swing and AWT

History:

1. AWT (1996)
2. Swing (1998)
3. JavaFX (2008)

- When Java was introduced, the GUI classes were bundled in a library known as the *Abstract Windows Toolkit (AWT)*.
- AWT is fine for developing simple graphical user interfaces, but not for developing comprehensive GUI projects.
- In addition, AWT is prone to platform-specific bugs.

JavaFX vs Swing and AWT

History:

1. AWT (1996)
2. Swing (1998)
3. JavaFX (2008)

AWT	SWING
Components are heavy-weight.	Light-weight components.
Native look and feel	Pluggable look and feel
Does not have MVC	Supports MVC
Not available in AWT.	Swing has many advanced features like JTabel, Jtabbedpane.
Components are platform dependent.	Components are platform independent.
AWT components require java.awt package.	Swing components require javax.swing package.
Slower	Faster than AWT

- The AWT user-interface components were replaced by a more robust, versatile, and flexible library known as *Swing components*.
- Swing components depend less on the target platform and use less of the native GUI resource.
- Swing works only with desktop applications

JavaFX vs Swing and AWT

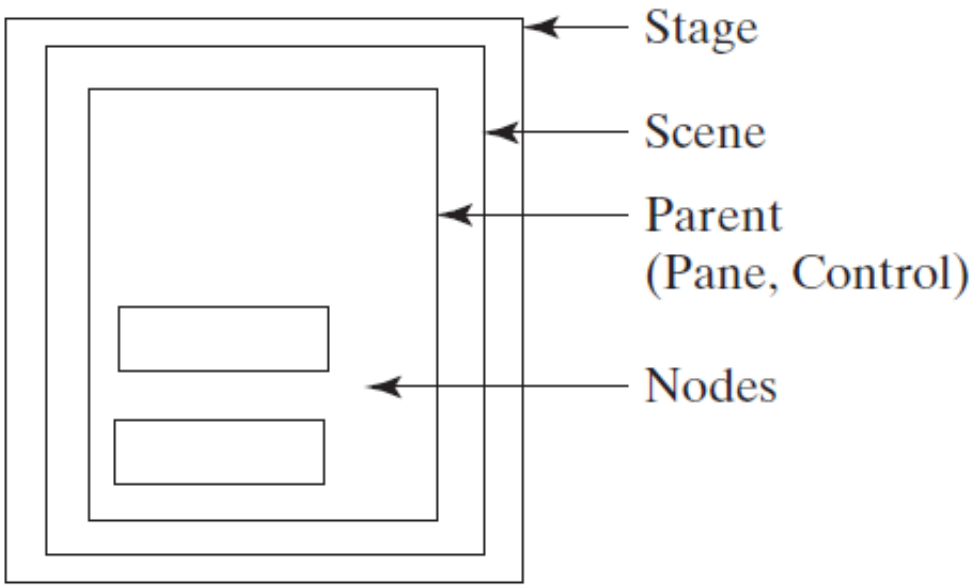
History:

1. AWT (1996)
2. Swing (1998)
3. JavaFX (2008)

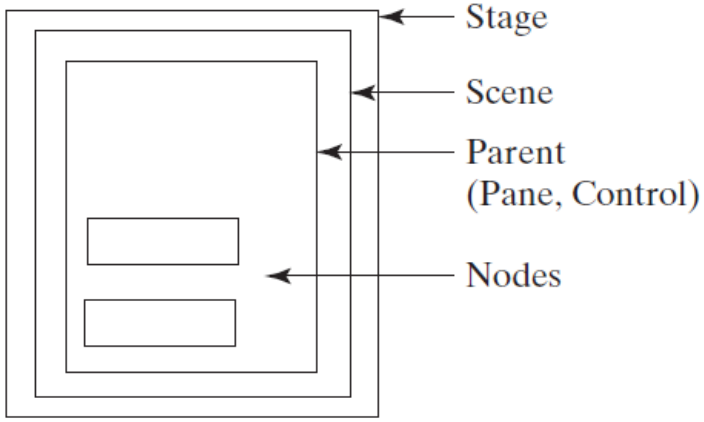
- With the release of Java 8, Swing is replaced by a completely new GUI platform known as *JavaFX*.
- With JavaFX you can control formatting with Cascading Style Sheets (CSS).
- JavaFX has more controls (collapsible, TitledPane and Accordion control)
- JavaFX has special effects (shadows, reflections, blurs, animations)

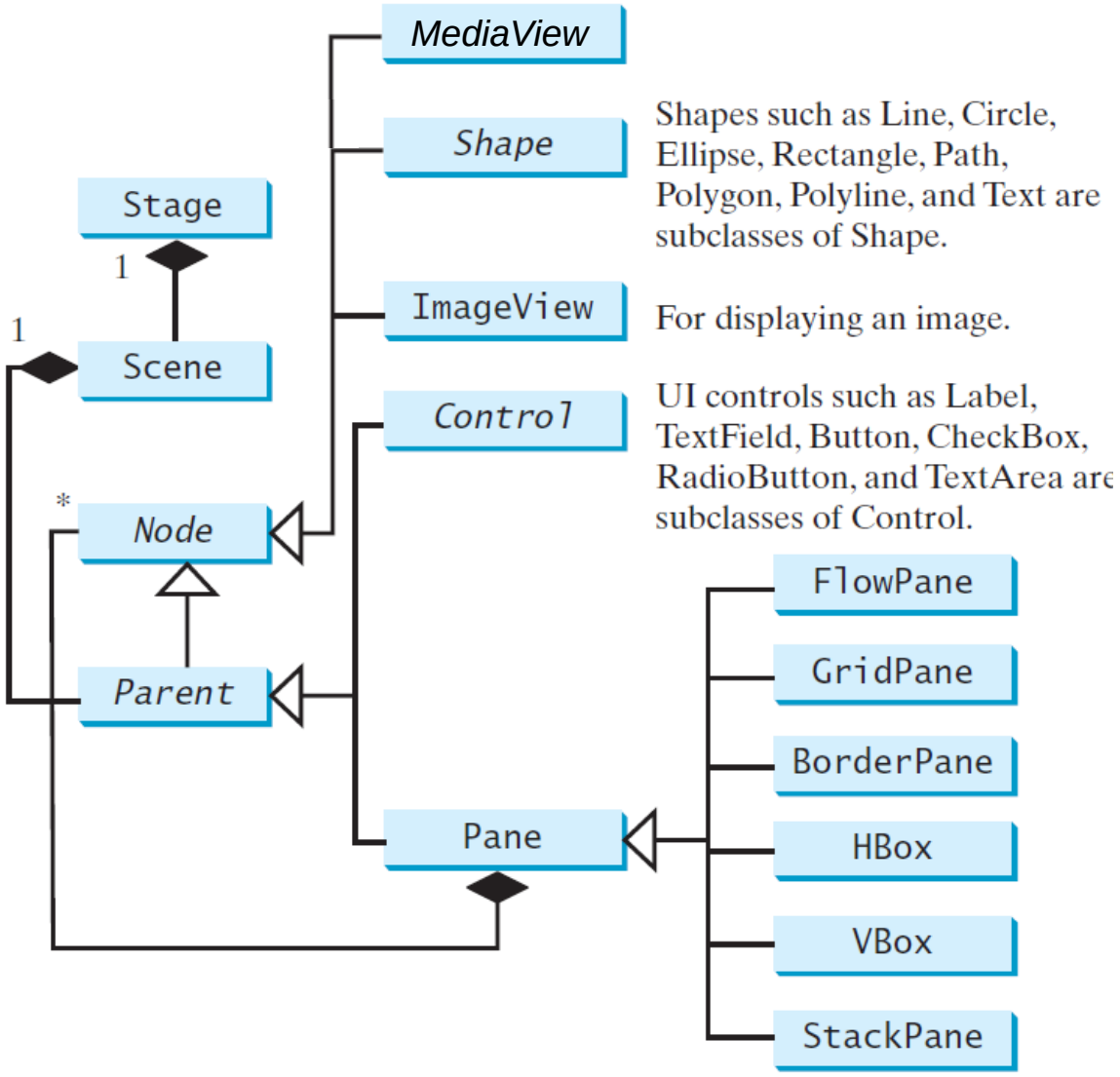
Basic Structure of JavaFX

- Application
- Override the start(Stage) method
- Stage, Scene, Node, and Parent



Panes, UI Controls, and Shapes





Stage

- A stage is the “location” where graphic elements will be displayed.
- It corresponds to a window in a desktop environment.
- A stage is created by instantiating a new instance of javafx.stage.Stage class.
- The “primary stage” (which corresponds to the main window of an application) is created automatically by the JavaFX framework when an application is launched, and is supplied as argument to the start() method of the Application class.

Scene

- A scene is a “container” for the set of graphic elements to be displayed on a given stage.
- The graphic elements are called “nodes” and are organized in a scene according to a hierarchical tree structure, with one root node and a set of (direct or indirect) children of the root node.
- This tree structure is referred to as the scene graph.
- The Java class representing a scene is javafx.scene.Scene.
- When a scene is assigned to a stage, all the scene contents (defined by the scene graph) are displayed on the stage.

Node and Parent

- Any graphic element is a subclass of the abstract class javafx.scene.Node.
- A scene graph must contain at least one node (the root node).
- Parent is the base class for all nodes that have children in the scene graph.

First Example

```
1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.control.Button;
4  import javafx.stage.Stage;
5
6  public class MyJavaFX extends Application {
7    @Override // Override the start method in the Application class
8    public void start(Stage primaryStage) {
9      // Create a scene and place a button in the scene
10     Button btOK = new Button("OK");
11     Scene scene = new Scene(btOK, 200, 250);
12     primaryStage.setTitle("MyJavaFX"); // Set the stage title
13     primaryStage.setScene(scene); // Place the scene in the stage
14     primaryStage.show(); // Display the stage
15   }
16
17   /**
18    * The main method is only needed for the IDE with limited
19    * JavaFX support. Not needed for running from the command line.
20    */
21   public static void main(String[] args) {
22     Application.launch(args);
23   }
24 }
```

Multi-Stage Example

```
1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.control.Button;
4  import javafx.stage.Stage;
5
6  public class MultipleStageDemo extends Application {
7    @Override // Override the start method in the Application class
8    public void start(Stage primaryStage) {
9      // Create a scene and place a button in the scene
10     Scene scene = new Scene(new Button("OK"), 200, 250);
11     primaryStage.setTitle("MyJavaFX"); // Set the stage title
12     primaryStage.setScene(scene); // Place the scene in the stage
13     primaryStage.show(); // Display the stage
14
15     Stage stage = new Stage(); // Create a new stage
16     stage.setTitle("Second Stage"); // Set the stage title
17     // Set a scene with a button in the stage
18     stage.setScene(new Scene(new Button("New Stage"), 100, 100));
19     stage.show(); // Display the stage
20   }
21 }
```

Chapter 14 JavaFX Basics

Panes, Color and Font Classes

Panes, UI Controls, and Shapes

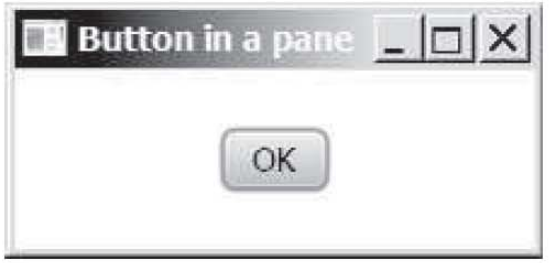
- A Pane is a container classes that is used for automatically laying out the nodes in the desired location and size.
- You place nodes inside a pane and then place the pane into a scene.
- A node is a visual component such as a shape, an image view, a UI control, or a pane.
- A *shape* refers to a text, line, circle, ellipse, rectangle, arc, polygon, polyline, etc.
- A *UI control* refers to a label, button, check box, radio button, text field, text area, etc.

Panes, UI Controls, and Shapes

- Scene can contain a Control or a Pane, but not a Shape or an ImageView, while a Pane can contain any subtype of Node.
- You can create a Scene using one of two ways:
- Constructor Scene(Parent, width, height) or,
- Constructor Scene(Parent)
- In the latter constructor the dimension of the scene is automatically decided by JRM.
- Every subclass of Node has a no-arg constructor for creating a default node.

Button Simple Example

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.stage.Stage;
import javafx.scene.layout.StackPane;

public class ButtonInPane extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create a scene and place a button in the scene
    StackPane pane = new StackPane();
    pane.getChildren().add(new Button("OK"));
    Scene scene = new Scene(pane, 200, 50);
    primaryStage.setTitle("Button in a pane"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }
}
```

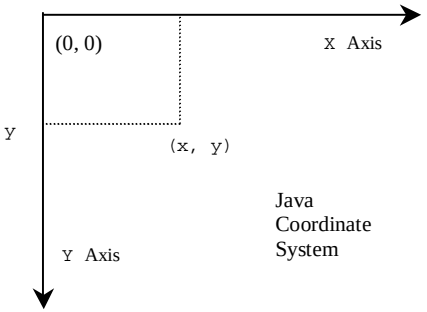


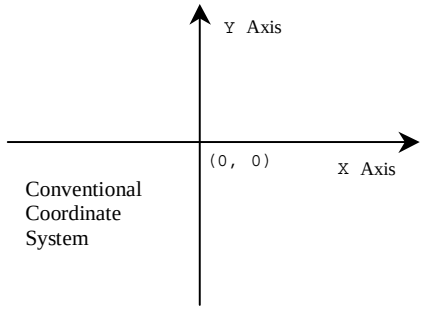
Button Simple Example

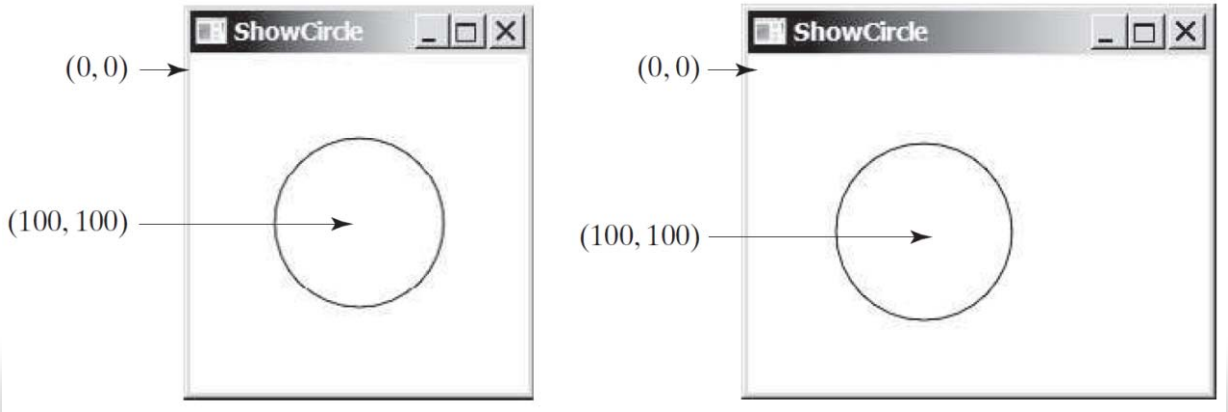
- The getChildren() method returns an instance of javafx.collections.ObservableList of Nodes.
- ObservableList behaves very much like an ArrayList for storing a collection of elements.
- Invoking add(e) adds an element to the list.
- The StackPane places the nodes in the center of the pane on top of each other.
- Here, there is only one node in the pane.
- The StackPane respects a node's default size. So you see the button displayed in its default size.

Another Button Example

```
public class ShowCircle extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create a circle and set its properties
    Circle circle = new Circle();
    circle.setCenterX(100);
    circle.setCenterY(100);
    circle.setRadius(50);
    circle.setStroke(Color.BLACK);
    circle.setFill(Color.WHITE);

    // Create a pane to hold the circle
    Pane pane = new Pane();
    pane.getChildren().add(circle);

    // Create a scene and place it in the stage
    Scene scene = new Scene(pane, 200, 200);
    primaryStage.setTitle("ShowCircle"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }
}
```







Common Properties and Methods for Nodes

- Nodes share many common properties. The most important among these is the **style** property.
- The style properties in JavaFX are called *JavaFX cascading style sheets (CSS)*.
- Each node has its own style properties.
- In JavaFX, a style property is defined with a prefix –fx-.
- The syntax for setting a style is **styleName:value**.
- Multiple style properties for a node can be set together separated by semicolon (;).

Common Properties and Methods for Nodes

- For example, the following statement
  circle.setStyle**("-fx-stroke: black**; **-fx-fill: red**;");

  sets two JavaFX CSS properties for a circle.
- This statement is equivalent to the following two statements.
  - ✓ circle.setStroke(Color.BLACK);
  - ✓ circle.setFill(Color.RED);
- If an incorrect JavaFX CSS is used, your program will still compile and run, but the style is ignored.

The Color Class

The getter methods for property values are provided in the class, but omitted in the UML diagram for brevity.

javafx.scene.paint.Color

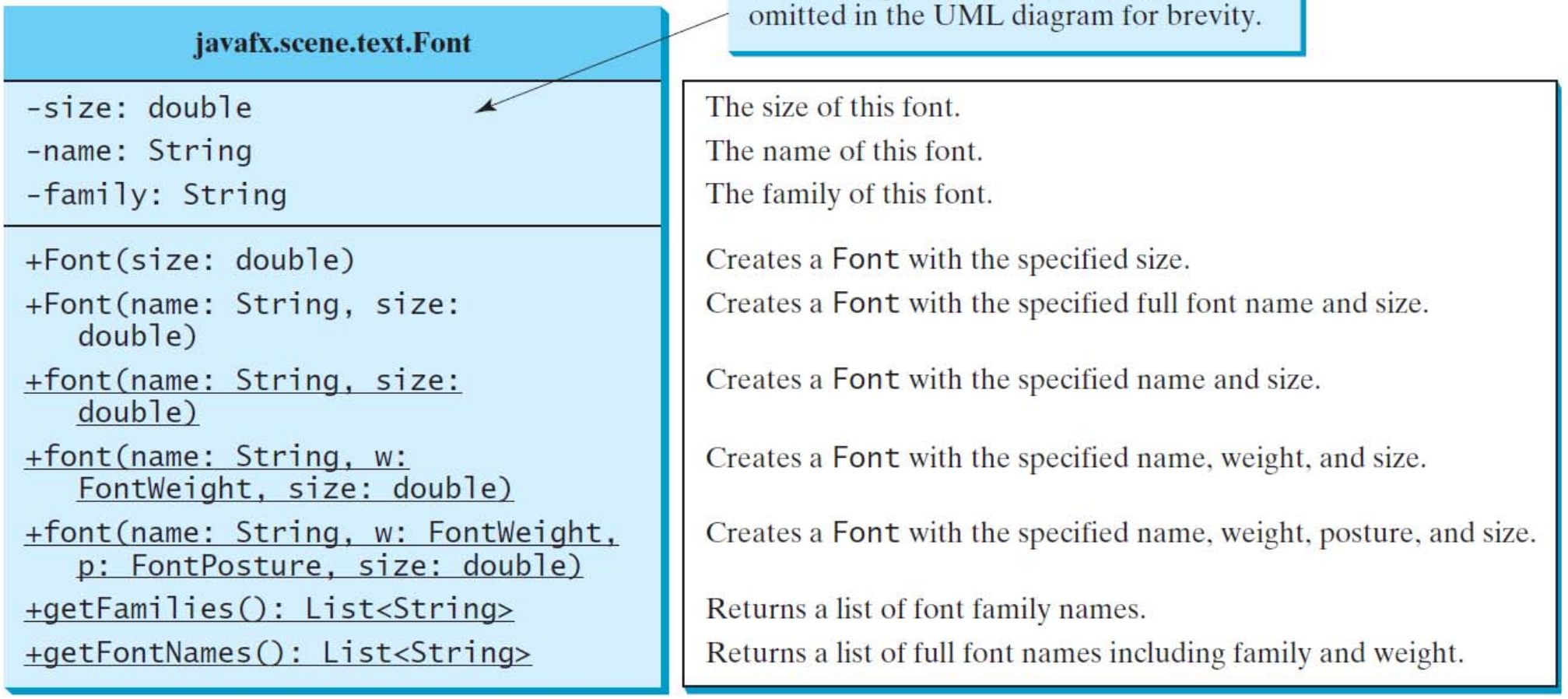
-red: double	The red value of this Color (between 0.0 and 1.0).
-green: double	The green value of this Color (between 0.0 and 1.0).
-blue: double	The blue value of this Color (between 0.0 and 1.0).
-opacity: double	The opacity of this Color (between 0.0 and 1.0).
+Color(r: double, g: double, b: double, opacity: double)	Creates a Color with the specified red, green, blue, and opacity values.
+brighter(): Color	Creates a Color that is a brighter version of this Color.
+darker(): Color	Creates a Color that is a darker version of this Color.
+color(r: double, g: double, b: double): Color	Creates an opaque Color with the specified red, green, and blue values.
+color(r: double, g: double, b: double, opacity: double): Color	Creates a Color with the specified red, green, blue, and opacity values.
+rgb(r: int, g: int, b: int): Color	Creates a Color with the specified red, green, and blue values in the range from 0 to 255.
+rgb(r: int, g: int, b: int, opacity: double): Color	Creates a Color with the specified red, green, and blue values in the range from 0 to 255 and a given opacity.

The Color Class

- JavaFX defines the abstract Paint class for painting a node.
- The javafx.scene.paint.Color is a subclass of Paint.
- The Color class is immutable. Once a Color object is created, its properties cannot be changed.
- The brighter() method returns a new Color with a larger red, green, and blue values.
- The darker() method returns a new Color with a smaller red, green, and blue values.
- Built-in static colors can be used, for example: circle.setFill(**Color.RED**);

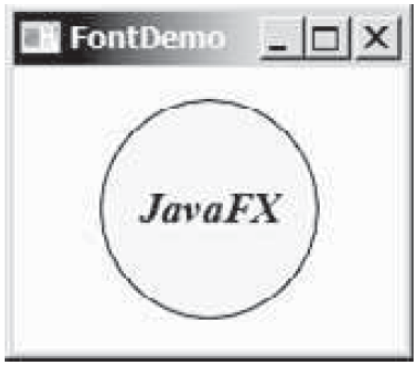
The Font Class

The getter methods for property values are provided in the class, but omitted in the UML diagram for brevity.



Font Example

```
public class FontDemo extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create a pane to hold the circle
    Pane pane = new StackPane();

    // Create a circle and set its properties
    Circle circle = new Circle();
    circle.setRadius(50);
    circle.setStroke(Color.BLACK);
    circle.setFill(new Color(0.5, 0.5, 0.5, 0.1));
    pane.getChildren().add(circle); // Add circle to the pane

    // Create a label and set its properties
    Label label = new Label("JavaFX");
    label.setFont(Font.font("Times New Roman",
      FontWeight.BOLD, FontPosture.ITALIC, 20));
    pane.getChildren().add(label);

    // Create a scene and place it in the stage
    Scene scene = new Scene(pane);
    primaryStage.setTitle("FontDemo"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }
}
```



The Font Class

- A **StackPane** places the nodes in the center and nodes are placed on top of each other.
- As you resize the window, the circle and label are displayed in the center of the window, because the circle and label are placed in the stack pane.
- Stack pane automatically places nodes in the center of the pane.
- A **Font** object is immutable. Once a **Font** object is created, its properties cannot be changed.

Layout Panes

JavaFX provides many types of panes for organizing nodes in a container.

Class	*Description*
`Pane`	Base class for layout panes. It contains the `getChildren()` method for returning a list of nodes in the pane.
`StackPane`	Places the nodes on top of each other in the center of the pane.
`FlowPane`	Places the nodes row-by-row horizontally or column-by-column vertically.
`GridPane`	Places the nodes in the cells in a two-dimensional grid.
`BorderPane`	Places the nodes in the top, right, bottom, left, and center regions.
`HBox`	Places the nodes in a single row.
`VBox`	Places the nodes in a single column.

Layout Panes

- **Pane** is the base class for all specialized panes.
- Each pane contains a list for holding nodes in the pane.
- This list is an instance of **ObservableList**, which can be obtained using pane's **getChildren**() method.
- You can use the **add**(node) method to add an element to the list, or
- use **addAll**(node1, node2, ...) to add a variable number of nodes to the pane.
- A node can be added to only one pane and only one time. Adding a node to a pane multiple times or to different panes will cause a runtime error.

FlowPane

- Arranges the nodes in the pane horizontally from left to right or vertically from top to bottom in the order in which they were added.
- When one row or one column is filled, a new row or column is started.
- Data fields **alignment**, **orientation**, **hgap**, and **vgap** are Object properties.
- Each Object property in JavaFX has a getter method (e.g., getHgap()) that returns its value, a setter method (e.g., sethGap(double)) for setting a value, and a getter method that returns the property itself (e.g., hGapProperty()).

FlowPane

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

javafx.scene.layout.FlowPane

```
-alignment: ObjectProperty<Pos>
-orientation:
   ObjectProperty<Orientation>
-hgap: DoubleProperty
-vgap: DoubleProperty
```

```
+FlowPane()
+FlowPane(hgap: double, vgap:
   double)
+FlowPane(orientation:
   ObjectProperty<Orientation>)
+FlowPane(orientation:
   ObjectProperty<Orientation>,
   hgap: double, vgap: double
```

The overall alignment of the content in this pane (default: `Pos.LEFT`).
The orientation in this pane (default: `Orientation.HORIZONTAL`).

The horizontal gap between the nodes (default: 0).
The vertical gap between the nodes (default: 0).

Creates a default `FlowPane`.
Creates a `FlowPane` with a specified horizontal and vertical gap.

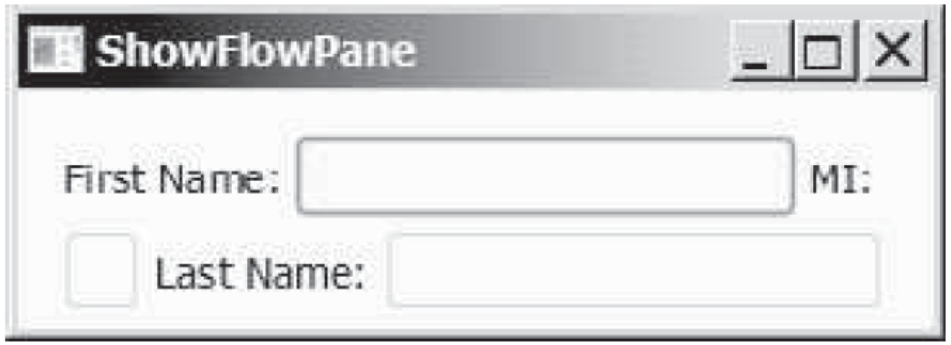
Creates a `FlowPane` with a specified orientation.

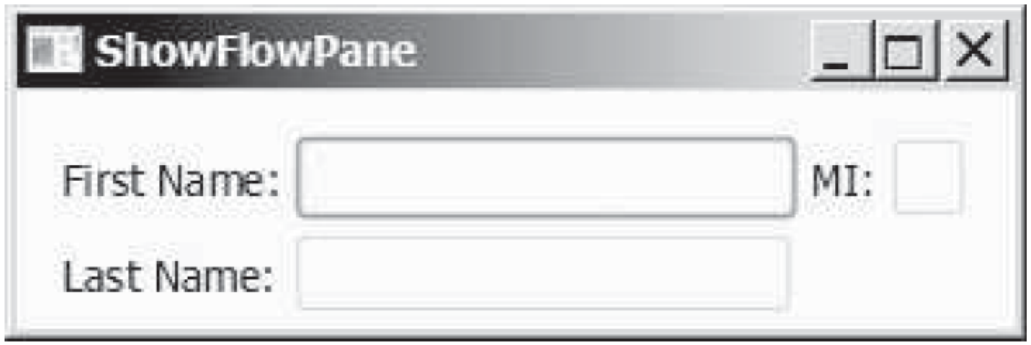
Creates a `FlowPane` with a specified orientation, horizontal gap and vertical gap.

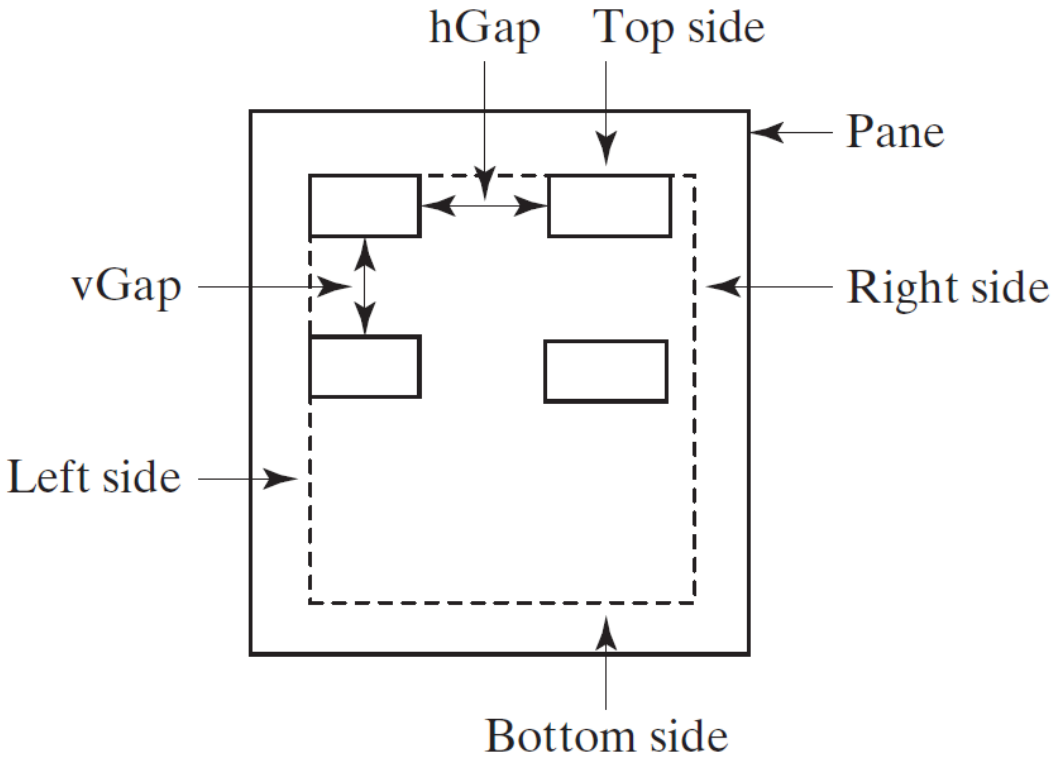
FlowPane Example

```
public class ShowFlowPane extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create a pane and set its properties
    FlowPane pane = new FlowPane();
    pane.setPadding(new Insets(11, 12, 13, 14));
    pane.setHgap(5);
    pane.setVgap(5);

    // Place nodes in the pane
    pane.getChildren().addAll(new Label("First Name:"),
      new TextField(), new Label("MI:"));
    TextField tfMi = new TextField();
    tfMi.setPrefColumnCount(1);
    pane.getChildren().addAll(tfMi, new Label("Last Name:"),
      new TextField());

    // Create a scene and place it in the stage
    Scene scene = new Scene(pane, 200, 250);
    primaryStage.setTitle("ShowFlowPane"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }
}
```

FlowPane Example







GridPane

- A GridPane arranges nodes in a grid (matrix) formation.
- The nodes are placed in the specified column and row indices.
- The column and row indexes start from 0.
- Not every cell in the grid needs to be filled.
- To remove a node from a GridPane, use pane.getChildren().remove(node).
- To remove all nodes, use pane.getChildren().removeAll().

GridPane

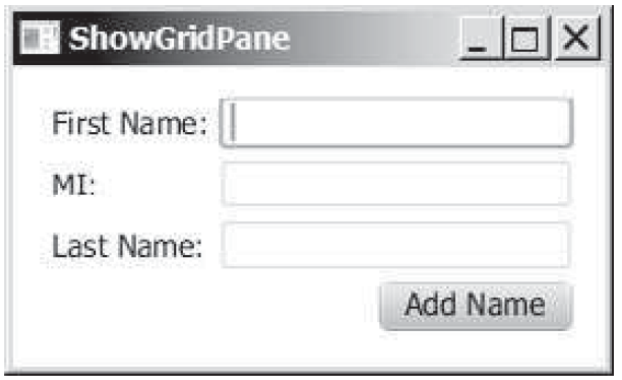
The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

javafx.scene.layout.GridPane

-alignment: ObjectProperty<Pos>	The overall alignment of the content in this pane (default: Pos.LEFT).
-gridLinesVisible: BooleanProperty	Is the grid line visible? (default: false)
-hgap: DoubleProperty	The horizontal gap between the nodes (default: 0).
-vgap: DoubleProperty	The vertical gap between the nodes (default: 0).
+GridPane()	Creates a GridPane.
+add(child: Node, columnIndex: int, rowIndex: int): void	Adds a node to the specified column and row.
+addColumn(columnIndex: int, children: Node...): void	Adds multiple nodes to the specified column.
+addRow(rowIndex: int, children: Node...): void	Adds multiple nodes to the specified row.
+getColumnIndex(child: Node): int	Returns the column index for the specified node.
+setColumnIndex(child: Node, columnIndex: int): void	Sets a node to a new column. This method repositions the node.
+getRowIndex(child:Node): int	Returns the row index for the specified node.
+setRowIndex(child: Node, rowIndex: int): void	Sets a node to a new row. This method repositions the node.
+setHalighnment(child: Node, value: HPos): void	Sets the horizontal alignment for the child in the cell.
+setValighnment(child: Node, value: VPos): void	Sets the vertical alignment for the child in the cell.

GridPane

```
public class ShowGridPane extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create a pane and set its properties
    GridPane pane = new GridPane();
    pane.setAlignment(Pos.CENTER);
    pane.setPadding(new Insets(11.5, 12.5, 13.5, 14.5));
    pane.setHgap(5.5);
    pane.setVgap(5.5);

    // Place nodes in the pane
    pane.add(new Label("First Name:"), 0, 0);
    pane.add(new TextField(), 1, 0);
    pane.add(new Label("MI:"), 0, 1);
    pane.add(new TextField(), 1, 1);
    pane.add(new Label("Last Name:"), 0, 2);
    pane.add(new TextField(), 1, 2);
    Button btAdd = new Button("Add Name");
    pane.add(btAdd, 1, 3);
    GridPane.setHalignment(btAdd, HPos.RIGHT);

    // Create a scene and place it in the stage
    Scene scene = new Scene(pane);
    primaryStage.setTitle("ShowGridPane"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }
}
```



BorderPane

- A BorderPane can place nodes in five regions: top, bottom, left, right, and center, using the setTop(node), setBottom(node), setLeft(node), setRight(node), and setCenter(node) methods.
- Note that a pane is a node. So a pane can be added into another pane.
- To remove a node from the top region, invoke **setTop**(null).
- If a region is not occupied, no space will be allocated for this region.

BorderPane

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

javafx.scene.layout.BorderPane

```
-top: ObjectProperty<Node>
-right: ObjectProperty<Node>
-bottom: ObjectProperty<Node>
-left: ObjectProperty<Node>
-center: ObjectProperty<Node>
```

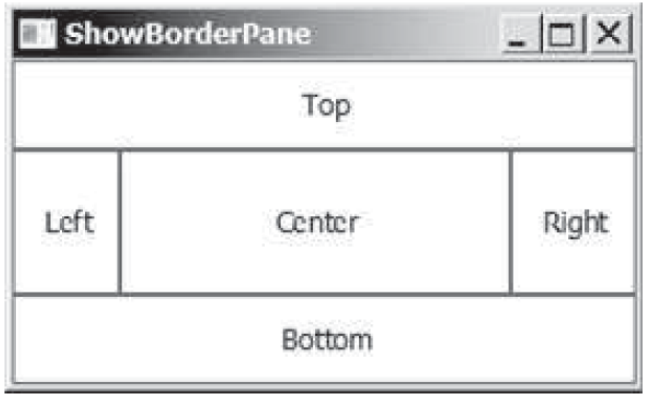
```
+BorderPane()
+setAlignment(child: Node, pos: Pos)
```

The node placed in the top region (default: `null`).
The node placed in the right region (default: `null`).
The node placed in the bottom region (default: `null`).
The node placed in the left region (default: `null`).
The node placed in the center region (default: `null`).

Creates a `BorderPane`.
Sets the alignment of the node in the `BorderPane`.

BorderPane

```
public class ShowBorderPane extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create a border pane
    BorderPane pane = new BorderPane();

    // Place nodes in the pane
    pane.setTop(new CustomPane("Top"));
    pane.setRight(new CustomPane("Right"));
    pane.setBottom(new CustomPane("Bottom"));
    pane.setLeft(new CustomPane("Left"));
    pane.setCenter(new CustomPane("Center"));

    // Create a scene and place it in the stage
    Scene scene = new Scene(pane);
    primaryStage.setTitle("ShowBorderPane"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }
}

// Define a custom pane to hold a label in the center of the pane
class CustomPane extends StackPane {
  public CustomPane(String title) {
    getChildren().add(new Label(title));
    setStyle("-fx-border-color: red");
    setPadding(new Insets(11.5, 12.5, 13.5, 14.5));
  }
}
```

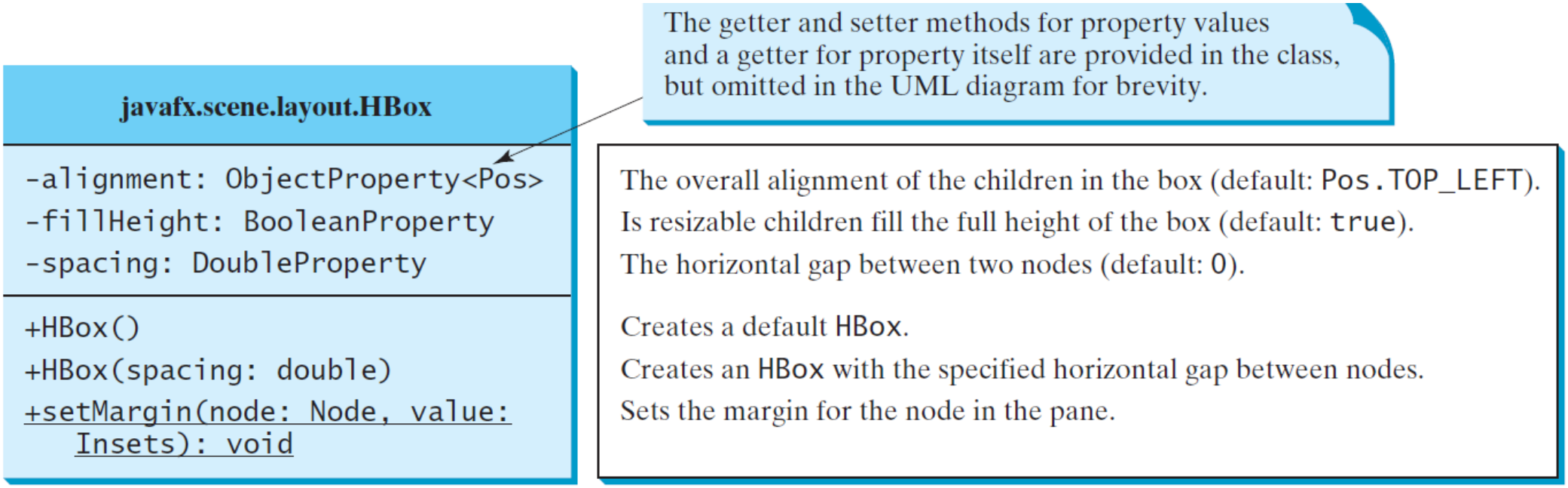


Chapter 14 JavaFX Basics

HBox and Vbox
Shapes

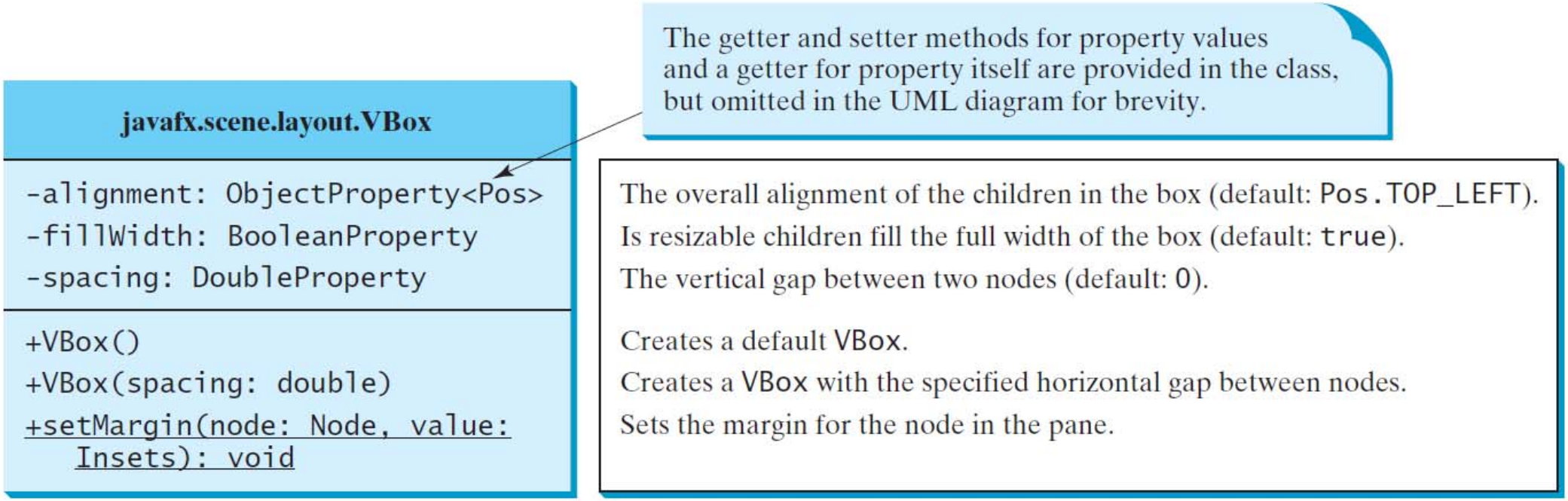
Hbox

- An **HBox** lays out its children in a single horizontal row.



VBox

- A **VBox** lays out its children in a single vertical column.



Hbox and Vbox Example

```
public class ShowHBoxVBox extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create a border pane
    BorderPane pane = new BorderPane();

    // Place nodes in the pane
    pane.setTop(getHBox());
    pane.setLeft(getVBox());

    // Create a scene and place it in the stage
    Scene scene = new Scene(pane);
    primaryStage.setTitle("ShowHBoxVBox"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }

  private HBox getHBox() {
    HBox hBox = new HBox(15);
    hBox.setPadding(new Insets(15, 15, 15, 15));
    hBox.setStyle("-fx-background-color: gold");
    hBox.getChildren().add(new Button("Computer Science"));
    hBox.getChildren().add(new Button("Chemistry"));
    ImageView imageView = new ImageView(new Image("image/us.gif"));
    hBox.getChildren().add(imageView);
    return hBox;
  }
```

Hbox and Vbox Example (continued)

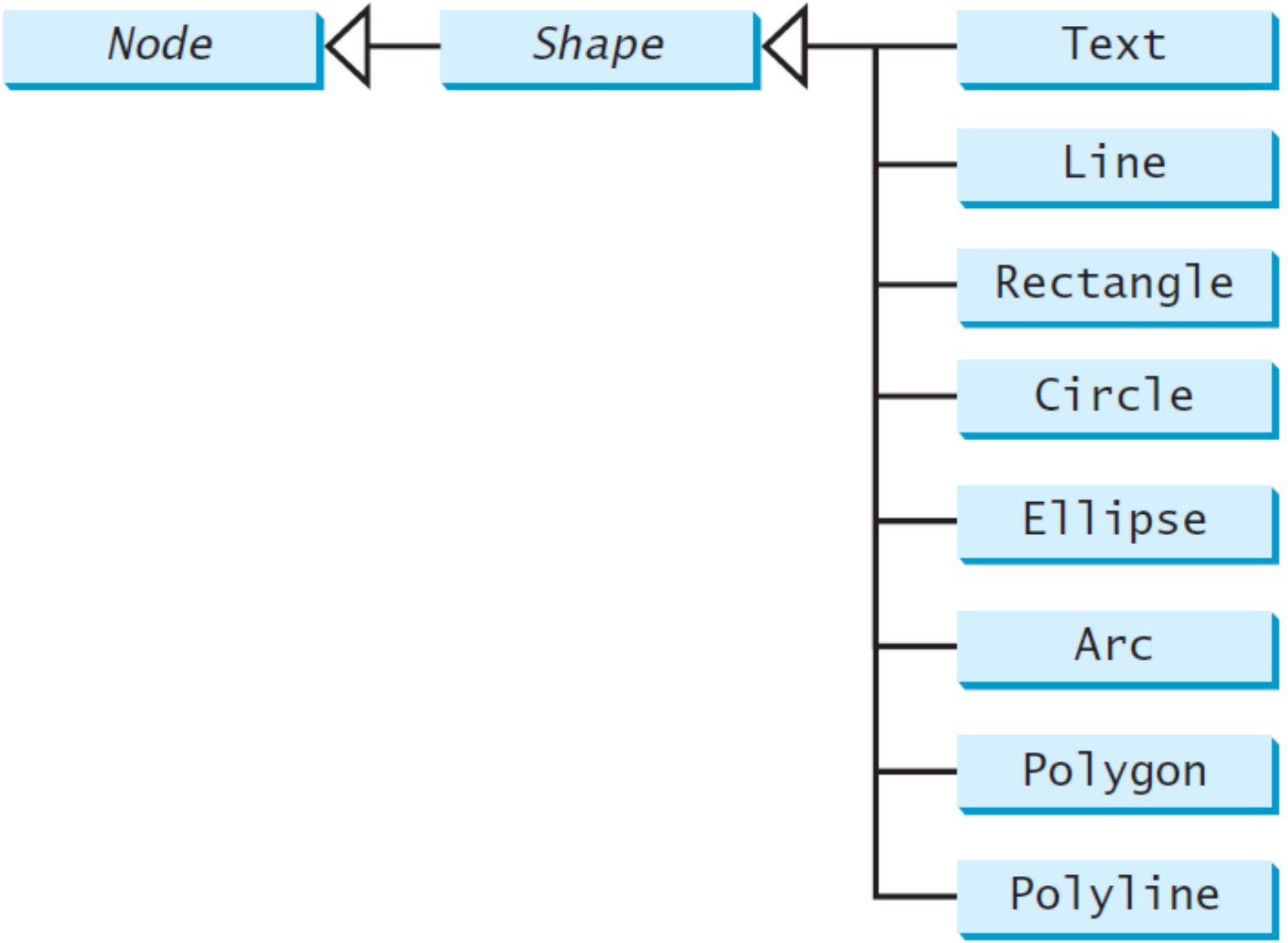
```
private VBox getVBox() {
  VBox vBox = new VBox(15);
  vBox.setPadding(new Insets(15, 5, 5, 5));
  vBox.getChildren().add(new Label("Courses"));

  Label[] courses = {new Label("CSCI 1301"), new Label("CSCI 1302"),
      new Label("CSCI 2410"), new Label("CSCI 3720")};

  for (Label course: courses) {
    VBox.setMargin(course, new Insets(0, 0, 0, 15));
    vBox.getChildren().add(course);
  }

  return vBox;
  }
}
```

Shapes

- JavaFX provides many shape classes for drawing texts, lines, circles, rectangles, ellipses, arcs, polygons, and polylines.
- The **Shape** class is the abstract base class that defines the common properties for all shapes.
- Among them are the **fill**, **stroke**, and **strokeWidth** properties.
- The **fill** property specifies a color that fills the interior of a shape.
- The **stroke** property specifies a color that is used to draw the outline of a shape.
- The **strokeWidth** property specifies the width of the outline of a shape.

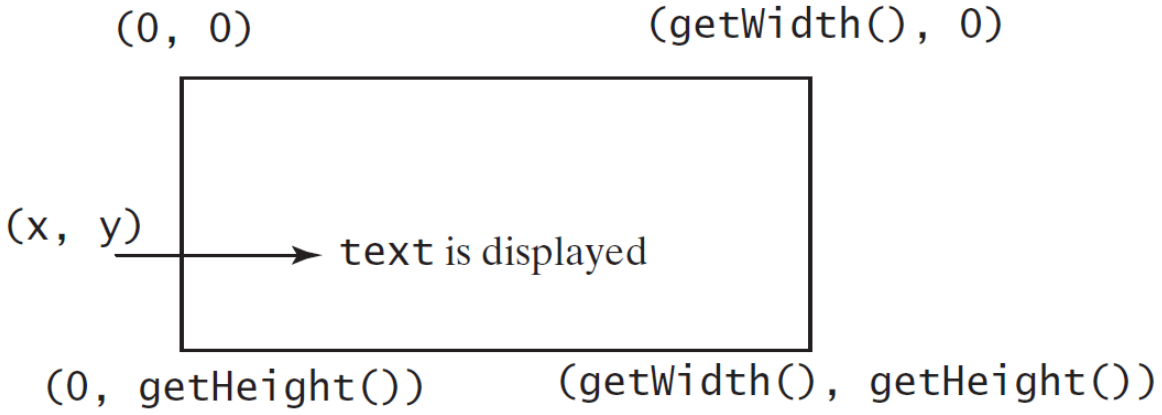
Shapes

Subclasses of abstract class **Shape**



Text

- The **Text** class defines a node that displays a string at a starting point (**x**, **y**).
- A **Text** object is usually placed in a pane.
- The pane's upper-left corner point is (**0**, **0**) and the bottom-right point is (**pane.getWidth**(), **pane.getHeight**()).
- A string may be displayed in multiple lines separated by **n**.



(a) Text(x, y, text)

Text

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

javafx.scene.text.Text

```
-text: StringProperty
-x: DoubleProperty
-y: DoubleProperty
-underline: BooleanProperty
-strikethrough: BooleanProperty
-font: ObjectProperty<Font>
```

```
+Text()
+Text(text: String)
+Text(x: double, y: double,
   text: String)
```

Defines the text to be displayed.
Defines the x-coordinate of text (default 0).
Defines the y-coordinate of text (default 0).
Defines if each line has an underline below it (default `false`).
Defines if each line has a line through it (default `false`).
Defines the font for the text.

Creates an empty Text.
Creates a Text with the specified text.
Creates a Text with the specified x-, y-coordinates and text.

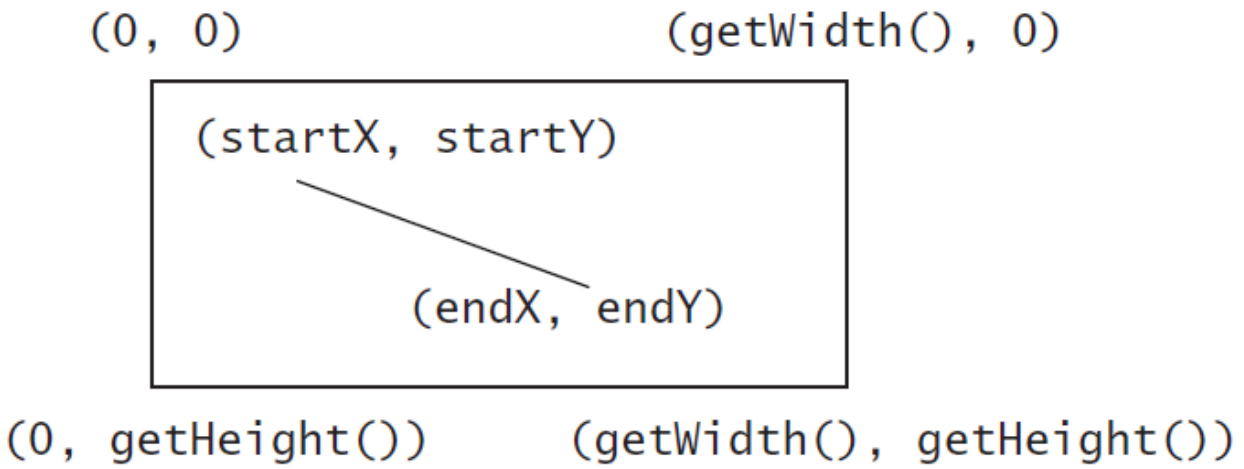
Text Example

```
public class ShowText extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create a pane to hold the texts
    Pane pane = new Pane();
    pane.setPadding(new Insets(5, 5, 5, 5));
    Text text1 = new Text(20, 20, "Programming is fun");
    text1.setFont(Font.font("Courier", FontWeight.BOLD,
      FontPosture.ITALIC, 15));
    pane.getChildren().add(text1);

    Text text2 = new Text(60, 60, "Programming is fun\nDisplay text");
    pane.getChildren().add(text2);

    Text text3 = new Text(10, 100, "Programming is fun\nDisplay text");
    text3.setFill(Color.RED);
    text3.setUnderline(true);
    text3.setStrikethrough(true);
    pane.getChildren().add(text3);

    // Create a scene and place it in the stage
    Scene scene = new Scene(pane);
    primaryStage.setTitle("ShowText"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }
}
```

(b) *Three* Text *objects are displayed*

Line

- The **Line** class is used to connect two points with four parameters startX, startY, endX, and endY.



The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

javafx.scene.shape.Line

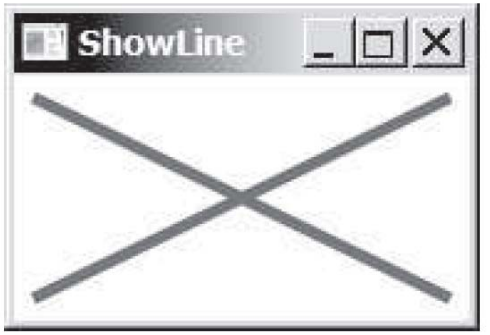
```
-startX: DoubleProperty
-startY: DoubleProperty
-endX: DoubleProperty
-endY: DoubleProperty
```

```
+Line()
+Line(startX: double, startY:
   double, endX: double, endY:
   double)
```

The x-coordinate of the start point.
The y-coordinate of the start point.
The x-coordinate of the end point.
The y-coordinate of the end point.

Creates an empty `Line`.
Creates a `Line` with the specified starting and ending points.

Line Example



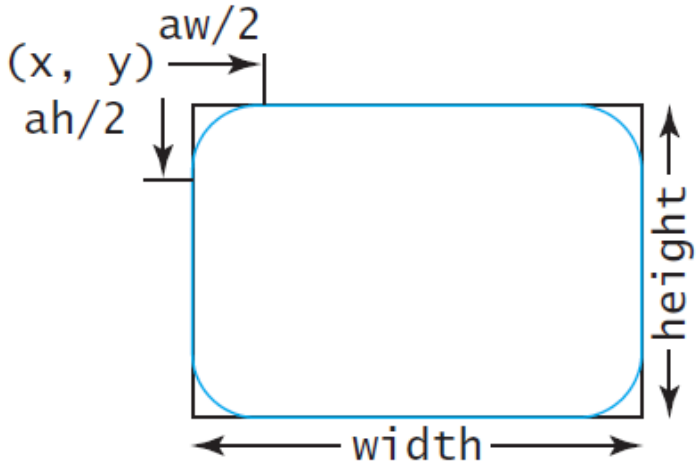
(b) Two lines are displayed across the pane.

```
class LinePane extends Pane {
  public LinePane() {
    Line line1 = new Line(10, 10, 10, 10);
    line1.endXProperty().bind(widthProperty().subtract(10));
    line1.endYProperty().bind(heightProperty().subtract(10));
    line1.setStrokeWidth(5);
    line1.setStroke(Color.GREEN);
    getChildren().add(line1);

    Line line2 = new Line(10, 10, 10, 10);
    line2.startXProperty().bind(widthProperty().subtract(10));
    line2.endYProperty().bind(heightProperty().subtract(10));
    line2.setStrokeWidth(5);
    line2.setStroke(Color.GREEN);
    getChildren().add(line2);
  }
}
```

Rectangle

- A rectangle is defined by the parameters **x**, **y**, **width**, **height**, **arcWidth**, and **arcHeight**.
- The rectangle's upper-left corner point is at (x, y)
- Parameter **arcWidth** is the horizontal diameter of the arcs at the corner, and **arcHeight** is the vertical diameter of the arcs at the corner.
- In class Rectangle, the fill color is black and the stroke color is white by default.
- If the method **setFill(null**) of **Rectangle** is called, the rectangle is not filled with a color.

Rectangle



(a) `Rectangle(x, y, w, h)`

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

javafx.scene.shape.Rectangle

```
-x: DoubleProperty
-y:DoubleProperty
-width: DoubleProperty
-height: DoubleProperty
-arcWidth: DoubleProperty

-arcHeight: DoubleProperty
```

```
+Rectangle()
+Rectanlge(x: double, y:
   double, width: double,
   height: double)
```

The x-coordinate of the upper-left corner of the rectangle (default 0).

The y-coordinate of the upper-left corner of the rectangle (default 0).

The width of the rectangle (default: 0).

The height of the rectangle (default: 0).

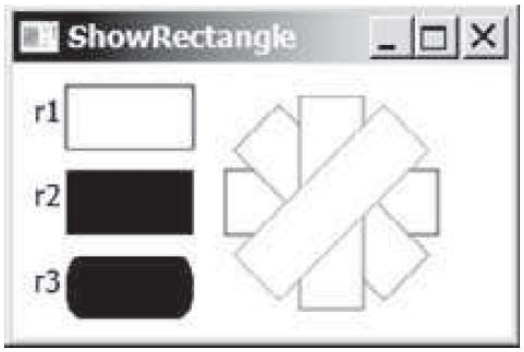
The `arcWidth` of the rectangle (default: 0). `arcWidth` is the horizontal diameter of the arcs at the corner (see Figure 14.31a).

The `arcHeight` of the rectangle (default: 0). `arcHeight` is the vertical diameter of the arcs at the corner (see Figure 14.31a).

Creates an empty `Rectangle`.

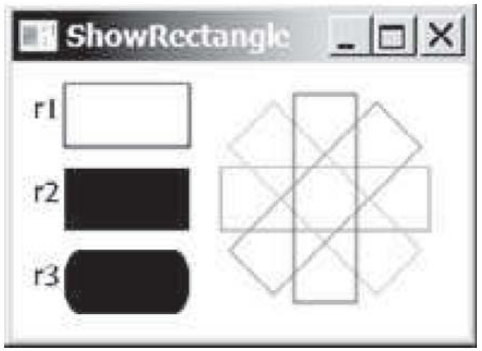
Creates a `Rectangle` with the specified upper-left corner point, width, and height.

Rectangle Example



(b) Multiple rectangles are displayed

```
r.setFill(null);
```

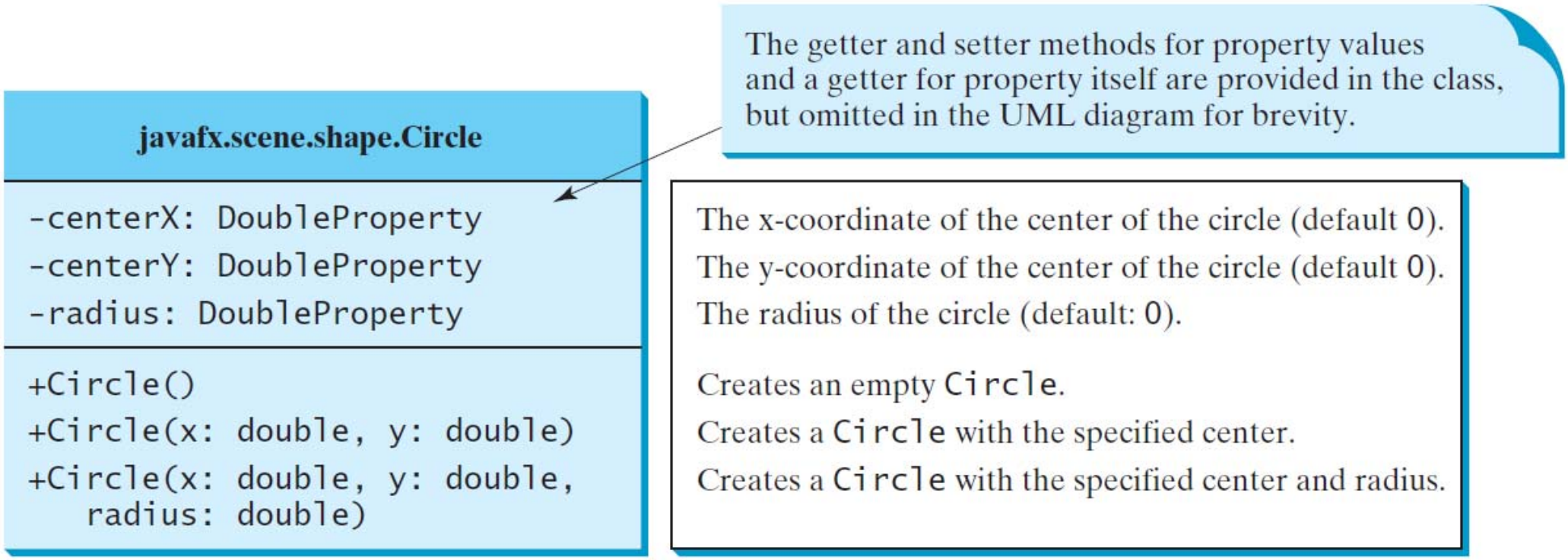


(c) Transparent rectangles are displayed

```
Pane pane = new Pane();

// Create rectangles and add to pane
Rectangle r1 = new Rectangle(25, 10, 60, 30);
r1.setStroke(Color.BLACK);
r1.setFill(Color.WHITE);
pane.getChildren().add(new Text(10, 27, "r1"));
pane.getChildren().add(r1);

Rectangle r2 = new Rectangle(25, 50, 60, 30);
pane.getChildren().add(new Text(10, 67, "r2"));
pane.getChildren().add(r2);

Rectangle r3 = new Rectangle(25, 90, 60, 30);
r3.setArcWidth(15);
r3.setArcHeight(25);
pane.getChildren().add(new Text(10, 107, "r3"));
pane.getChildren().add(r3);

for (int i = 0; i < 4; i++) {
  Rectangle r = new Rectangle(100, 50, 100, 30);
  r.setRotate(i * 360 / 8);
  r.setStroke(Color.color(Math.random(), Math.random(),
      Math.random()));
  r.setFill(Color.WHITE);
  pane.getChildren().add(r);
}

// Create a scene and place it in the stage
Scene scene = new Scene(pane, 250, 150);
primaryStage.setTitle("ShowRectangle"); // Set the stage title
primaryStage.setScene(scene); // Place the scene in the stage
primaryStage.show(); // Display the stage
  }
}
```

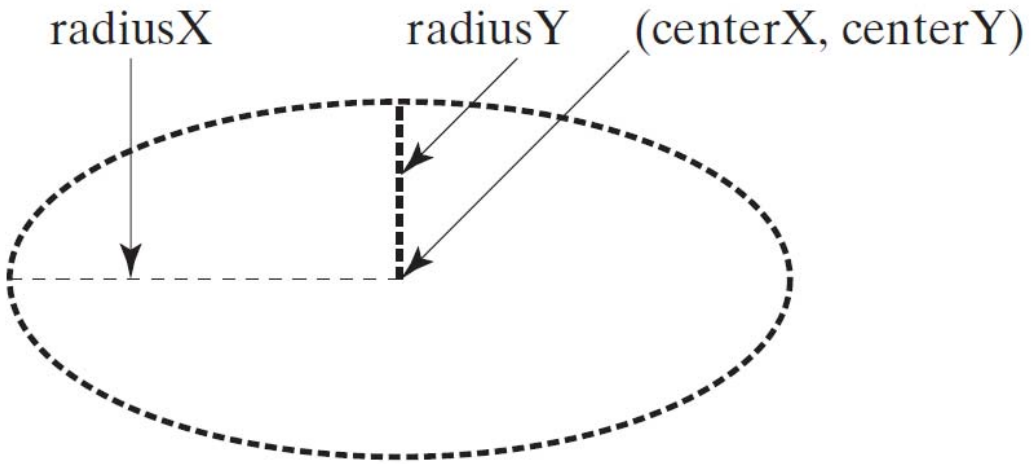
Circle

A circle is defined by its parameters **centerX**, **centerY**, and **radius**.



Ellipse

An ellipse is defined by its parameters **centerX**, **centerY**, **radiusX**, and **radiusY**



The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

javafx.scene.shape.Ellipse

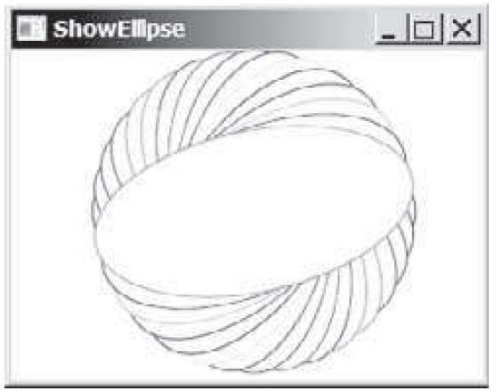
```
-centerX: DoubleProperty
-centerY: DoubleProperty
-radiusX: DoubleProperty
-radiusY: DoubleProperty
```

```
+Ellipse()
+Ellipse(x: double, y: double)
+Ellipse(x: double, y: double,
   radiusX: double, radiusY:
   double)
```

The x-coordinate of the center of the ellipse (default 0).
The y-coordinate of the center of the ellipse (default 0).
The horizontal radius of the ellipse (default: 0).
The vertical radius of the ellipse (default: 0).

Creates an empty `Ellipse`.
Creates an `Ellipse` with the specified center.
Creates an `Ellipse` with the specified center and radiuses.

Ellipse Example



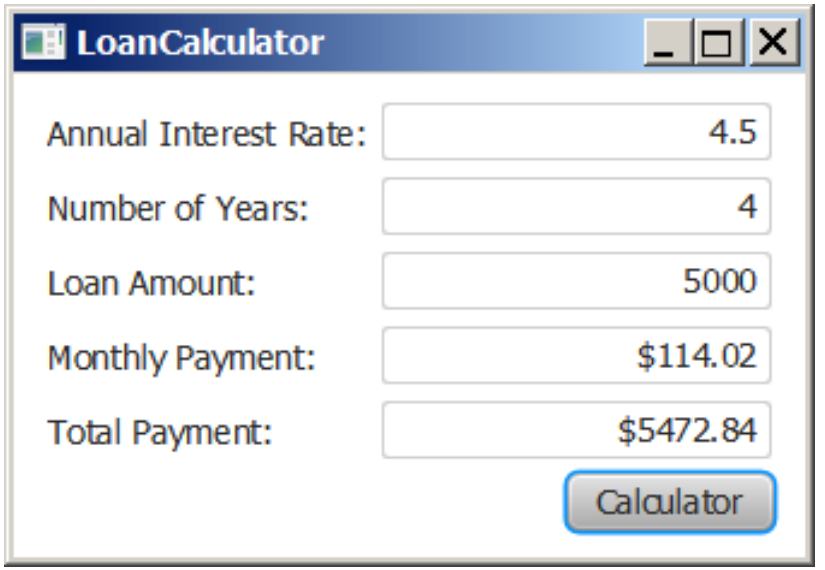
(b) Multiple ellipses are displayed.

```
public void start(Stage primaryStage) {
  // Create a pane
  Pane pane = new Pane();

  for (int i = 0; i < 16; i++) {
    // Create an ellipse and add it to pane
    Ellipse e1 = new Ellipse(150, 100, 100, 50);
    e1.setStroke(Color.color(Math.random(), Math.random(),
      Math.random()));
    e1.setFill(Color.WHITE);
    e1.setRotate(i * 180 / 16);
    pane.getChildren().add(e1);
  }

  // Create a scene and place it in the stage
  Scene scene = new Scene(pane, 300, 200);
  primaryStage.setTitle("ShowEllipse"); // Set the stage title
  primaryStage.setScene(scene); // Place the scene in the stage
  primaryStage.show(); // Display the stage
}
```

Chapter 15 Event-Driven Programming and Animations

Motivations

Suppose you want to write a GUI program that lets the user enter a loan amount, annual interest rate, and number of years and click the *Compute Payment* button to obtain the monthly payment and total payment. How do you accomplish the task? You have to use *event-driven programming* to write the code to respond to the button-clicking event.



Procedural vs. Event-Driven Programming

- *Procedural programming* is executed in procedural order. User cannot interrupt the execution flow and cannot interact with the program.

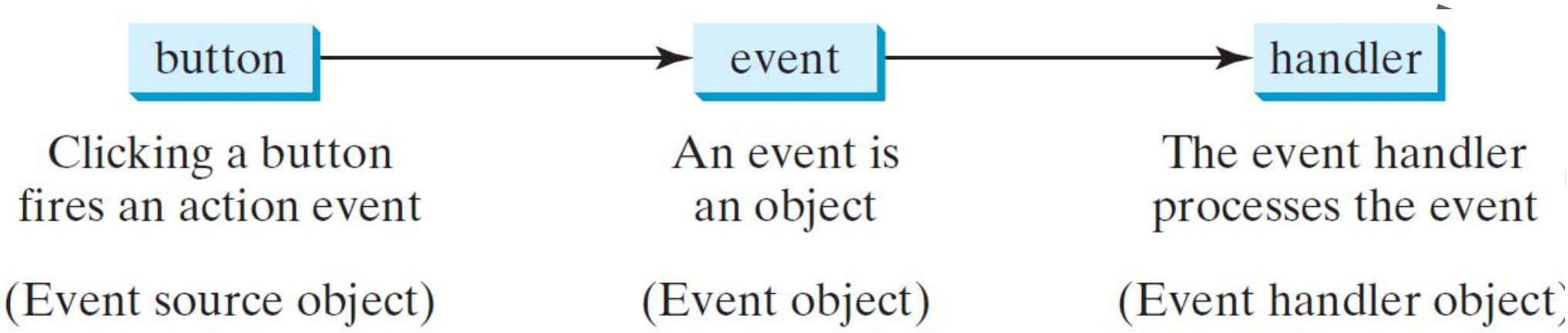
- In event-driven programming, code is executed upon activation of events. An event is initialized by a user's action and the corresponding response to the event is executed by the program.

Handling GUI Events

- An event is an object created from an event source (e.g., button).
- An event can be defined as a signal to the program that something has happened.
- Events are triggered by external user actions, such as mouse movements, mouse clicks, and keystrokes.

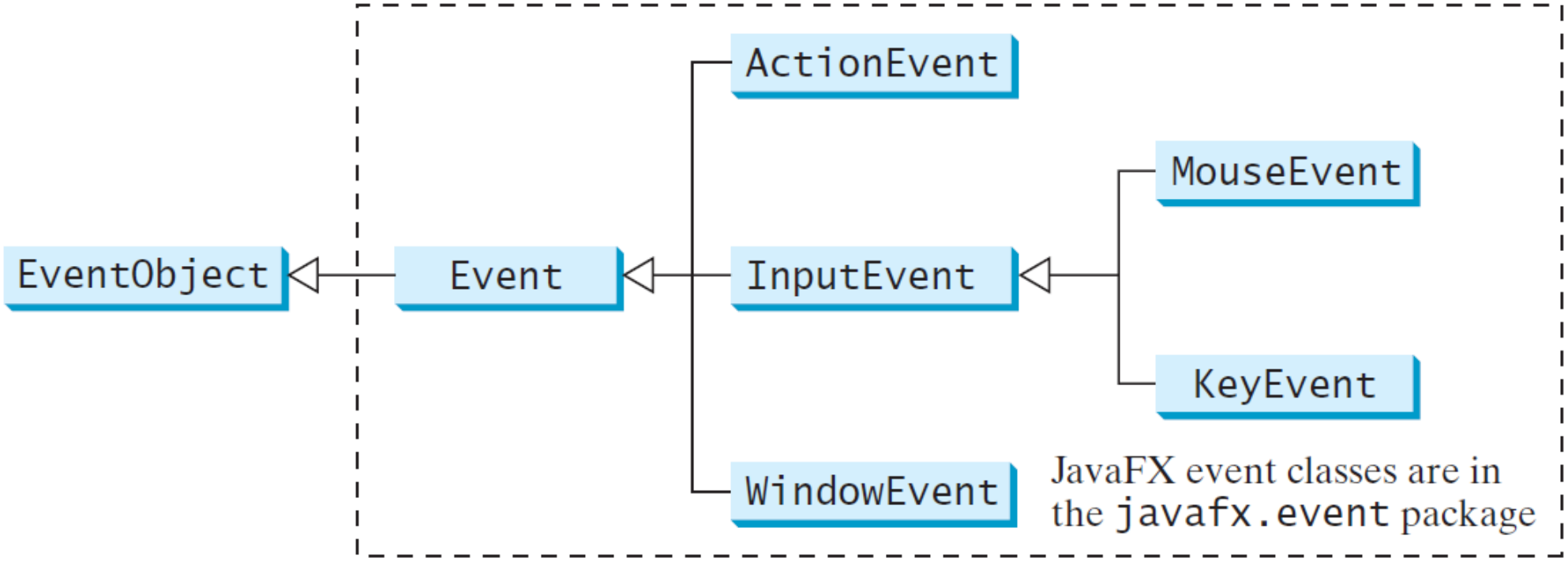
Handling GUI Events

- Firing an event means to create an event object and pass it to an object of type event handler that handles; i.e., executes the response to the event.
  - Source object (e.g., button)
  - Listener object contains a method for processing the event.



Event Classes

- An event is an instance of an **Event** class.
- The root class of the Java event classes is **java.util.EventObject**.
- The root class of the JavaFX event classes is **javafx.event.Event**.



Event Information

- An event object contains whatever properties that are relevant to the event.
- You can identify the source object of the event using the **getSource**() instance method in the **EventObject** class.
- The subclasses of **EventObject** deal with special types of events, such as button actions, window events, mouse movements, and keystrokes.
- For example, when clicking a button, the button creates and fires an ActionEvent. Here, the button is an event source object, and an ActionEvent is the event object fired by the source object

Selected User Actions and Handlers

User Action	*Source Object*	*Event Type Fired*	*Event Registration Method*
Click a button	`Button`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Press Enter in a text field	`TextField`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Check or uncheck	`RadioButton`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Check or uncheck	`CheckBox`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Select a new item	`ComboBox`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Mouse pressed	`Node, Scene`	`MouseEvent`	`setOnMousePressed(EventHandler<MouseEvent>)`
Mouse released			`setOnMouseReleased(EventHandler<MouseEvent>)`
Mouse clicked			`setOnMouseClicked(EventHandler<MouseEvent>)`
Mouse entered			`setOnMouseEntered(EventHandler<MouseEvent>)`
Mouse exited			`setOnMouseExited(EventHandler<MouseEvent>)`
Mouse moved			`setOnMouseMoved(EventHandler<MouseEvent>)`
Mouse dragged			`setOnMouseDragged(EventHandler<MouseEvent>)`
Key pressed	`Node, Scene`	`KeyEvent`	`setOnKeyPressed(EventHandler<KeyEvent>)`
Key released			`setOnKeyReleased(EventHandler<KeyEvent>)`
Key typed			`setOnKeyTyped(EventHandler<KeyEvent>)`

The Delegation Model

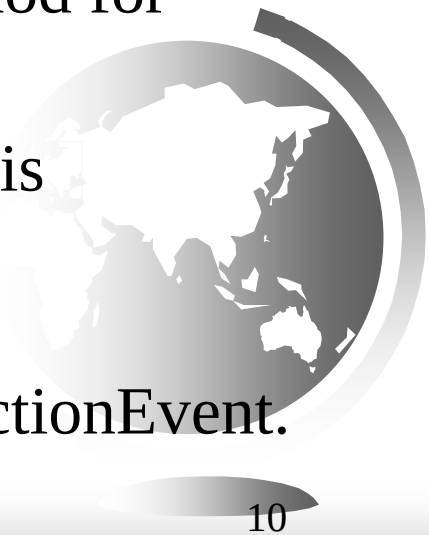
- Java uses a delegation-based model for event handling: a source object fires an event, and an object interested in the event handles it.
- The latter object is called an event handler or an event listener.

The Delegation Model

- For an object to be a handler for an event on a source object, two things are needed:

  1.

  - The handler object must be an instance of the corresponding event-handler interface to ensure that the handler has the correct method for processing the event.
  - JavaFX defines a unified handler interface EventHandler<T extends Event> for an event T.
  - The handler interface contains the handle(T e) method for processing the event.
  - For example, the handler interface for ActionEvent is EventHandler<ActionEvent>; each handler for ActionEvent should implement the handle(ActionEvent e) method for processing an ActionEvent.



The Delegation Model

- For an object to be a handler for an event on a source object, two things are needed (continued):

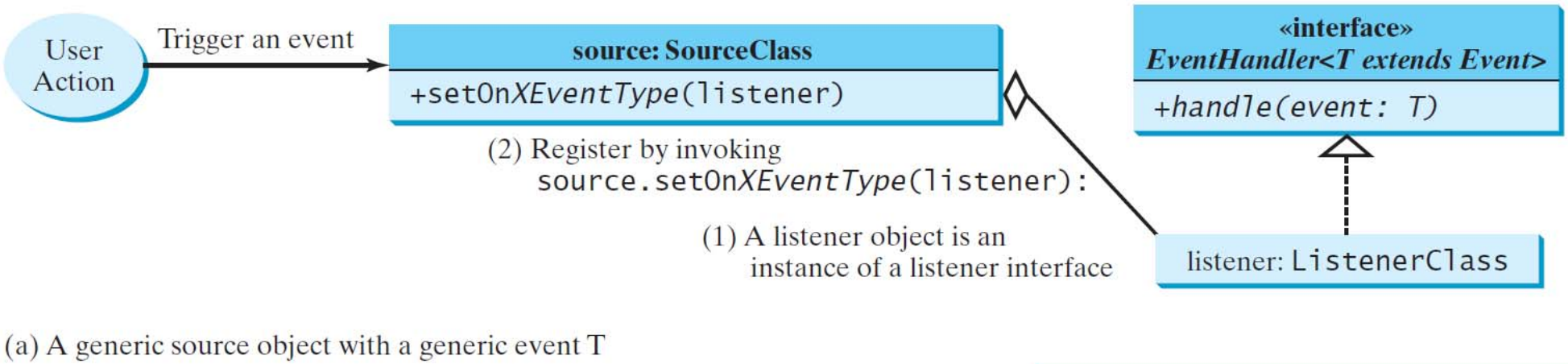
  2.

  - The handler object must be registered by the source object.
  - Registration methods depend on the event type.
  - For ActionEvent, the method is setOnAction().
  - For a mouse pressed event, the method is setOnMousePressed().
  - For a key pressed event, the method is setOnKeyPressed().

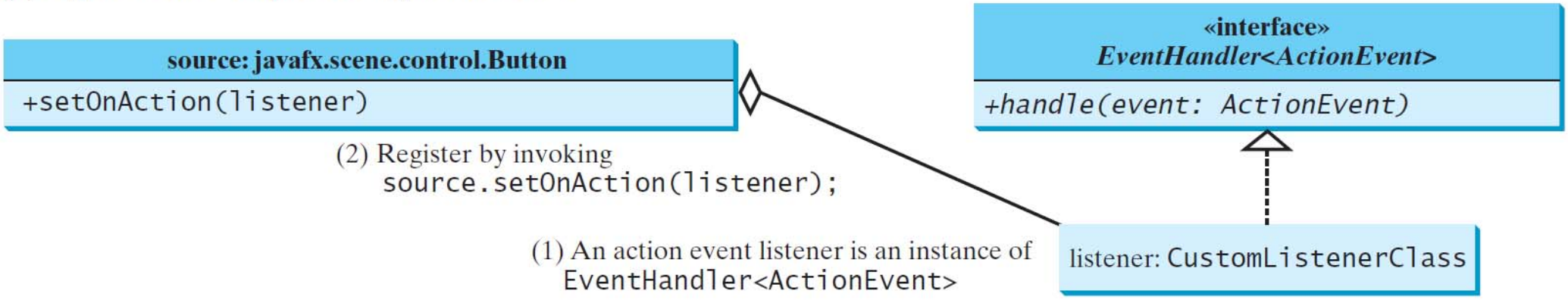
The Delegation Model

- For example, when you click a button, the Button object fires an **ActionEvent** and passes it to invoke the handler's **handle**(**ActionEvent e**) method to handle the event.
- The event object (**e**) contains information relevant to the event, which can be obtained using certain methods.
- For example, you can use **e.getSource**() to obtain the source object that fired the event.

The Delegation Model



(a) A generic source object with a generic event T



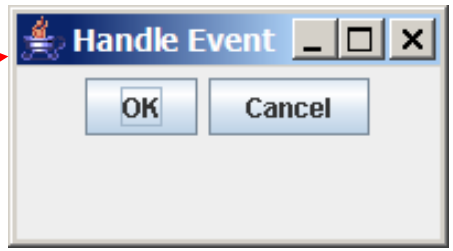
(b) A `Button` source object with an `ActionEvent`

animation

Trace Execution

```
public class HandleEvent extends Application {
  public void start(Stage primaryStage) {
    …
    OKHandlerClass handler1 = new OKHandlerClass();
    btOK.setOnAction(handler1);
    CancelHandlerClass handler2 = new CancelHandlerClass();
    btCancel.setOnAction(handler2);
    …
    primaryStage.show(); // Display the stage
  }
}
```

1. Start from the main method to create a window and display it

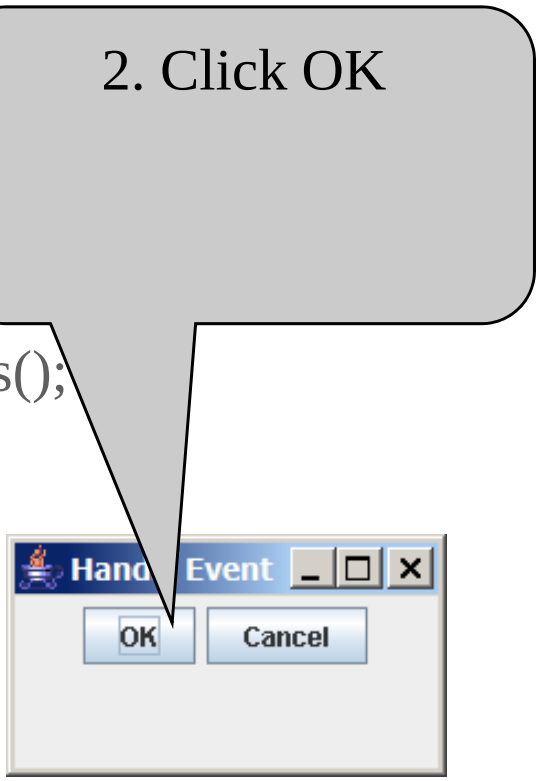



```
class OKHandlerClass implements EventHandler<ActionEvent> {
  @Override
  public void handle(ActionEvent e) {
    System.out.println("OK button clicked");
  }
}
```

animation

Trace Execution

```
public class HandleEvent extends Application {
  public void start(Stage primaryStage) {
    …
    OKHandlerClass handler1 = new OKHandlerClass();
    btOK.setOnAction(handler1);
    CancelHandlerClass handler2 = new CancelHandlerClass();
    btCancel.setOnAction(handler2);
    …
    primaryStage.show(); // Display the stage
  }
}
```



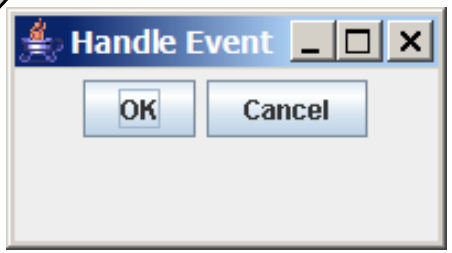

```
class OKHandlerClass implements EventHandler<ActionEvent> {
  @Override
  public void handle(ActionEvent e) {
    System.out.println("OK button clicked");
  }
}
```

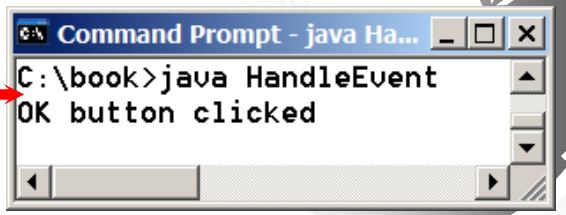
animation

Trace Execution

```
public class HandleEvent extends Application {
  public void start(Stage primaryStage) {
    …
    OKHandlerClass handler1 = new OKHandlerClass();
    btOK.setOnAction(handler1);
    CancelHandlerClass handler2 = new CancelHandlerClass
    btCancel.setOnAction(handler2);
    …
    primaryStage.show(); // Display the stage
  }
}

class OKHandlerClass implements EventHandler<ActionEvent> {
  @Override
  public void handle(ActionEvent e) {
    System.out.println("OK button clicked");
  }
}
```

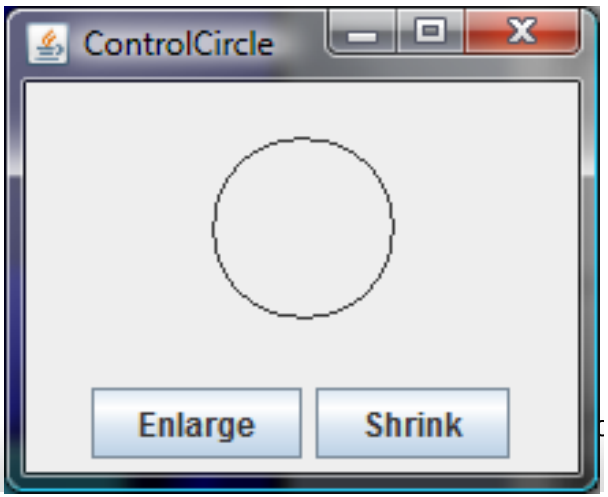
3. The JVM invokes the listener's handle method





Example: First Version for ControlCircle (no listeners)

Now let us consider to write a program that uses two buttons to control the size of a circle.




```
public class ControlCircleWithoutEventHandling extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    StackPane pane = new StackPane();
    Circle circle = new Circle(50);
    circle.setStroke(Color.BLACK);
    circle.setFill(Color.WHITE);
    pane.getChildren().add(circle);

    HBox hBox = new HBox();
    hBox.setSpacing(10);
    hBox.setAlignment(Pos.CENTER);
    Button btEnlarge = new Button("Enlarge");
    Button btShrink = new Button("Shrink");
    hBox.getChildren().add(btEnlarge);
    hBox.getChildren().add(btShrink);

    BorderPane borderPane = new BorderPane();
    borderPane.setCenter(pane);
    borderPane.setBottom(hBox);
    BorderPane.setAlignment(hBox, Pos.CENTER);

    // Create a scene and place it in the stage
    Scene scene = new Scene(borderPane, 200, 150);
    primaryStage.setTitle("ControlCircle"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }
}
```

Example ControlCircle

- How to use the buttons to enlarge or shrink the circle?
- When the Enlarge button is clicked, we want the circle to be repainted with a larger radius.
- When the Shrink button is clicked, we want the circle to be repainted with a smaller radius.
- How can we accomplish this?

Example ControlCircle

- First, we define a new class named **CirclePane** for displaying the circle in a pane.
- This new class displays a circle and provides the **enlarge** and **shrink** methods for increasing and decreasing the radius of the circle.
- It is a good strategy to design a class to model a circle pane with supporting methods so that these related methods along with the circle are coupled in one object.

Example ControlCircle

- Second, inside the ControlCircle class, we create a CirclePane object as a private data field in the ControlCircle class.
- The methods in the ControlCircle class can now access the CirclePane object through this data field.

Example ControlCircle

- Third, we define a handler class named **EnlargeHandler** that implements EventHandler<ActionEvent>.
- In order to make the reference variable circlePane accessible from the handle method, define EnlargeHandler as an inner class of the ControlCircle class.
- Inner classes are defined inside another class. We use an inner class here and will introduce it fully in the next lecture.

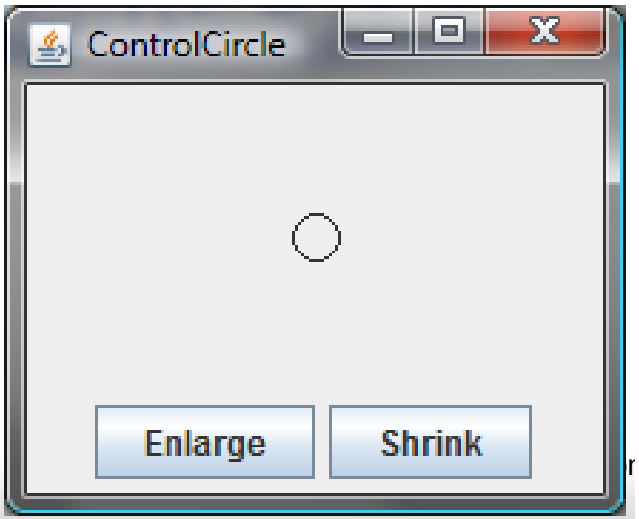
Example ControlCircle

- Finally, we register the handler for the **Enlarge** button and implement the handle method in EnlargeHandler to invoke circlePane.enlarge().

Example: Second Version for ControlCircle (with listener for Enlarge)

```
class CirclePane extends StackPane {
  private Circle circle = new Circle(50);

  public CirclePane() {
    getChildren().add(circle);
    circle.setStroke(Color.BLACK);
    circle.setFill(Color.WHITE);
  }

  public void enlarge() {
    circle.setRadius(circle.getRadius() + 2);
  }

  public void shrink() {
    circle.setRadius(circle.getRadius() > 2 ?
      circle.getRadius() - 2 : circle.getRadius());
  }
}
```

Example: Second Version for ControlCircle (with listener for Enlarge)




```
public class ControlCircle extends Application {
  private CirclePane circlePane = new CirclePane();

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Hold two buttons in an HBox
    HBox hBox = new HBox();
    hBox.setSpacing(10);
    hBox.setAlignment(Pos.CENTER);
    Button btEnlarge = new Button("Enlarge");
    Button btShrink = new Button("Shrink");
    hBox.getChildren().add(btEnlarge);
    hBox.getChildren().add(btShrink);

    // Create and register the handler
    btEnlarge.setOnAction(new EnlargeHandler());

    BorderPane borderPane = new BorderPane();
    borderPane.setCenter(circlePane);
    borderPane.setBottom(hBox);
    BorderPane.setAlignment(hBox, Pos.CENTER);

    // Create a scene and place it in the stage
    Scene scene = new Scene(borderPane, 200, 150);
    primaryStage.setTitle("ControlCircle"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }

  class EnlargeHandler implements EventHandler<ActionEvent> {
    @Override // Override the handle method
    public void handle(ActionEvent e) {
      circlePane.enlarge();
    }
  }
}
```

Chapter 15 Event-Driven Programming and Animations

Inner class handlers
Anonymous Inner class handlers
lambda expressions

Inner Classes

Inner class: A class is a member of another class.

Advantages: In some applications, you can use an inner class to make programs simple.

```
public class Test {
  ...
}

public class A {
  ...
}
```

(a)

```
public class Test {
  ...

  // Inner class
  public class A {
    ...
  }
}
```

(b)

```
// OuterClass.java: inner class demo
public class OuterClass {
  private int data;

  /** A method in the outer class */
  public void m() {
    // Do something
  }

  // An inner class
  class InnerClass {
    /** A method in the inner class */
    public void mi() {
      // Directly reference data and method
      // defined in its outer class
      data++;
      m();
    }
  }
}
```

(c)

Inner Classes (cont.)

- **Inner classes has the following features:**

1. An inner class is compiled into a class named: *OuterClassName*$*InnerClassName*.class.

For example, the inner class A in outer class Test is compiled into *Test$A*.class .

2. An inner class can reference the data and methods defined in the outer class in which it nests, so you do not need to pass the reference of the outer class to the constructor of the inner class.

Inner Classes (cont.)

3. An inner class can be declared public, protected, or private subject to the same visibility rules applied to a member of the class.

4. An inner class can be declared static. A static inner class can be accessed using the outer class name. A static inner class cannot access nonstatic members of the outer class.

5. If the inner class is public, you can create an object of the inner class from another class.

If the inner class is nonstatic

```
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

If the inner class is static

```
OuterClass.InnerClass innerObject = new OuterClass.InnerClass();
```

Inner Classes (cont.)

- A simple use of inner classes is to combine dependent classes into a primary class.
- This reduces the number of source files.
- It also makes class files easy to organize since they are all named with the primary class as the prefix.
- For example, rather than creating the two source files Test.java and A.java, you can merge class A into class Test and create just one source file, Test.java. The resulting class files are Test.class and Test$A.class.

Inner Class Handlers

An event handler class is designed specifically to create a handler object for a GUI component (e.g., a button). It will not be shared by other applications. So, it is appropriate to define the Event Handler class inside the Application class as an inner class.

Anonymous Inner Classes

- An anonymous inner class is an inner class without a name. It combines defining an inner class and creating an instance of the class into one step.

```
public void start(Stage primaryStage) {
  // Omitted

  btEnlarge.setOnAction(
    new EnlargeHandler());
}

class EnlargeHandler
    implements EventHandler<ActionEvent> {
  public void handle(ActionEvent e) {
    circlePane.enlarge();
  }
}
```

(a) Inner class EnlargeListener

```
public void start(Stage primaryStage) {
  // Omitted

  btEnlarge.setOnAction(
    new class EnlargeHandlner
      implements EventHandler<ActionEvent>() {
      public void handle(ActionEvent e) {
        circlePane.enlarge();
      }
    });
}
```

(b) Anonymous inner class

- The syntax for an anonymous inner class is:

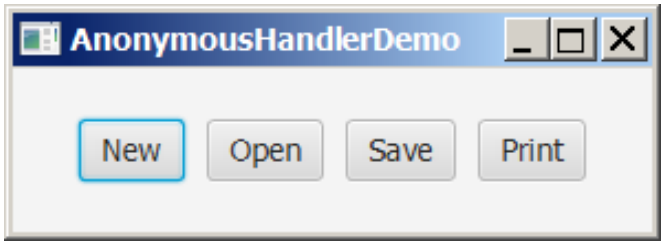
```
new SuperClassName/InterfaceName() {
  // Implement or override methods in superclass or interface

  // Other methods if necessary
}
```

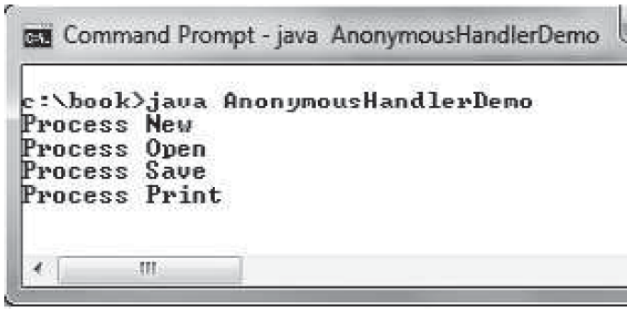
Anonymous Inner Classes

- An anonymous inner class must always extend a superclass or implement an interface, but it cannot have an explicit **extends** or **implements** clause.
- An anonymous inner class must implement all the abstract methods in the superclass or in the interface.
- An anonymous inner class always uses the **no-arg** constructor from its superclass to create an instance. If an anonymous inner class implements an interface, the constructor is **Object**().
- An anonymous inner class is compiled into a class named **OuterClassName$*n*.class**. For example, if the outer class **Test** has two anonymous inner classes, these two classes are compiled into **Test$1.class** and **Test$2.class**.

Anonymous Inner Classes Example

```
public class AnonymousHandlerDemo extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Hold two buttons in an HBox
    HBox hBox = new HBox();
    hBox.setSpacing(10);
    hBox.setAlignment(Pos.CENTER);
    Button btNew = new Button("New");
    Button btOpen = new Button("Open");
    Button btSave = new Button("Save");
    Button btPrint = new Button("Print");
    hBox.getChildren().addAll(btNew, btOpen, btSave, btPrint);

    // Create and register the handler
    btNew.setOnAction(new EventHandler<ActionEvent>() {
      @Override // Override the handle method
      public void handle(ActionEvent e) {
        System.out.println("Process New");
      }
    });

    btOpen.setOnAction(new EventHandler<ActionEvent>() {
      @Override // Override the handle method
      public void handle(ActionEvent e) {
        System.out.println("Process Open");
      }
    });
```





Anonymous Inner Classes Example

- Without using anonymous inner classes, we would have to create four separate classes.
- An anonymous handler works the same way as that of an inner class handler. The program is condensed using an anonymous inner class.
- The anonymous inner classes in this example are compiled into:
- **AnonymousHandlerDemo$1.class**, **AnonymousHandlerDemo$2.class**, **AnonymousHandlerDemo$3.class**, and **AnonymousHandlerDemo$4.class**.

Simplifying Event Handing Using Lambda Expressions

Lambda expression is a new feature in Java 8. Lambda expressions can be viewed as an anonymous class with a concise syntax. For example, the following code in (a) can be greatly simplified using a lambda expression in (b) in three lines.

```
btEnlarge.setOnAction(
  new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent e) {
      // Code for processing event e
    }
  }
});
```

(a) Anonymous inner class event handler

```
btEnlarge.setOnAction(e -> {
  // Code for processing event e
});
```

(b) Lambda expression event handler

Basic Syntax for a Lambda Expression

The basic syntax for a lambda expression is either

```
(type1 param1, type2 param2, ...) -> expression
```

or

```
(type1 param1, type2 param2, ...) -> { statements; }
```

The data type for a parameter may be explicitly declared or implicitly inferred by the compiler. The parentheses can be omitted if there is only one parameter without an explicit data type.

Lambda Expressions Example

```
public class LambdaHandlerDemo extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Hold two buttons in an HBox
    HBox hBox = new HBox();
    hBox.setSpacing(10);
    hBox.setAlignment(Pos.CENTER);
    Button btNew = new Button("New");
    Button btOpen = new Button("Open");
    Button btSave = new Button("Save");
    Button btPrint = new Button("Print");
    hBox.getChildren().addAll(btNew, btOpen, btSave, btPrint);

    // Create and register the handler
    btNew.setOnAction((ActionEvent e) -> {
      System.out.println("Process New");
    });

    btOpen.setOnAction((e) -> {
      System.out.println("Process Open");
    });

    btSave.setOnAction(e -> {
      System.out.println("Process Save");
    });

    btPrint.setOnAction(e -> System.out.println("Process Print"));
```

Lambda Expressions

- The compiler treats a lambda expression as if it is an object created from an anonymous inner class.
- In this case, the compiler understands that the object must be an instance of **EventHandler**<**ActionEvent**>.
- Since the **EventHandler** interface defines the handle method with a parameter of the **ActionEvent** type, the compiler automatically recognizes that **e** is a parameter of the **ActionEvent** type, and the statements are for the body of the **handle** method.

Lambda Expressions

- The **EventHandler** interface contains just one method. The statements in the lambda expression are all for that method.
- If it contains multiple methods, the compiler will not be able to compile the lambda expression.
- So, for the compiler to understand lambda expressions, the interface must contain exactly one **abstract** method.
- Such an interface is known as a *functional interface* or a *Single Abstract Method* (SAM) interface.

Mouse Events

- A **MouseEvent** is fired whenever a mouse button is pressed, released, clicked, moved, or dragged on a node or a scene.
- The **MouseEvent** object captures the event, such as the number of clicks associated with it, the location (the x- and y-coordinates) of the mouse, which mouse button was pressed …

Mouse Events

- Four constants—**PRIMARY**, **SECONDARY**, **MIDDLE**, and **NONE**—are defined in the MouseButton enumerator to indicate the left, right, middle, and none mouse buttons.

- We can use the **getButton**() method to detect which button is pressed. For example,

```
if (e.getButton() == MouseButton.SECONDARY)
```

checks that the right mouse button was pressed.

The `MouseEvent` Class

javafx.scene.input.MouseEvent	
`+getButton(): MouseButton`	Indicates which mouse button has been clicked.
`+getClickCount(): int`	Returns the number of mouse clicks associated with this event.
`+getX(): double`	Returns the *x*-coordinate of the mouse point in the event source node.
`+getY(): double`	Returns the *y*-coordinate of the mouse point in the event source node.
`+getSceneX(): double`	Returns the *x*-coordinate of the mouse point in the scene.
`+getSceneY(): double`	Returns the *y*-coordinate of the mouse point in the scene.
`+getScreenX(): double`	Returns the *x*-coordinate of the mouse point in the screen.
`+getScreenY(): double`	Returns the *y*-coordinate of the mouse point in the screen.
`+isAltDown(): boolean`	Returns true if the `Alt` key is pressed on this event.
`+isControlDown(): boolean`	Returns true if the `Control` key is pressed on this event.
`+isMetaDown(): boolean`	Returns true if the mouse `Meta` button is pressed on this event.
`+isShiftDown(): boolean`	Returns true if the `Shift` key is pressed on this event.

Mouse Events

User Action	*Source Object*	*Event Type Fired*	*Event Registration Method*
Click a button	`Button`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Press Enter in a text field	`TextField`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Check or uncheck	`RadioButton`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Check or uncheck	`CheckBox`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Select a new item	`ComboBox`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Mouse pressed	`Node, Scene`	`MouseEvent`	`setOnMousePressed(EventHandler<MouseEvent>)`
Mouse released			`setOnMouseReleased(EventHandler<MouseEvent>)`
Mouse clicked			`setOnMouseClicked(EventHandler<MouseEvent>)`
Mouse entered			`setOnMouseEntered(EventHandler<MouseEvent>)`
Mouse exited			`setOnMouseExited(EventHandler<MouseEvent>)`
Mouse moved			`setOnMouseMoved(EventHandler<MouseEvent>)`
Mouse dragged			`setOnMouseDragged(EventHandler<MouseEvent>)`
Key pressed	`Node, Scene`	`KeyEvent`	`setOnKeyPressed(EventHandler<KeyEvent>)`
Key released			`setOnKeyReleased(EventHandler<KeyEvent>)`
Key typed			`setOnKeyTyped(EventHandler<KeyEvent>)`

MouseEvent Example

```
public class MouseEventDemo extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create a pane and set its properties
    Pane pane = new Pane();
    Text text = new Text(20, 20, "Programming is fun");
    pane.getChildren().addAll(text);
    text.setOnMouseDragged(e -> {
      text.setX(e.getX());
      text.setY(e.getY());
    });

    // Create a scene and place it in the stage
    Scene scene = new Scene(pane, 300, 100);
    primaryStage.setTitle("MouseEventDemo"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }
}
```

Key Events

- A **KeyEvent** is fired whenever a key is pressed, released, or typed on a node or a scene.
- Key events enable the use of the keys to perform actions or to get input from the keyboard.
- The **KeyEvent** object describes the type of the event (key pressed, key released, or key typed) and the value of the key.

The KeyEvent Class

javafx.scene.input.KeyEvent	
+getCharacter(): String	Returns the character associated with the key in this event.
+getCode(): KeyCode	Returns the key code associated with the key in this event.
+getText(): String	Returns a string describing the key code.
+isAltDown(): boolean	Returns true if the Alt key is pressed on this event.
+isControlDown(): boolean	Returns true if the Control key is pressed on this event.
+isMetaDown(): boolean	Returns true if the mouse Meta button is pressed on this event.
+isShiftDown(): boolean	Returns true if the Shift key is pressed on this event.

Key Events

- Every key event has an associated code that is returned by the **getCode**() method in **KeyEvent**.
- The *key codes* are constants defined in the enumerator **KeyCode**.
- For the *key-pressed* and *key-released* events, **getCode**() returns the value as defined in the table, **getText**() returns a string that describes the key code, and **getCharacter**() returns an empty string.
- For the *key-typed* event, **getCode**() returns **UNDEFINED** and **getCharacter**() returns the Unicode character or a sequence of characters associated with the *key-typed* event.

The KeyCode Constants

Constant	*Description*	*Constant*	*Description*
HOME	The Home key	CONTROL	The Control key
END	The End key	SHIFT	The Shift key
PAGE_UP	The Page Up key	BACK_SPACE	The Backspace key
PAGE_DOWN	The Page Down key	CAPS	The Caps Lock key
UP	The up-arrow key	NUM_LOCK	The Num Lock key
DOWN	The down-arrow key	ENTER	The Enter key
LEFT	The left-arrow key	UNDEFINED	The keyCode unknown
RIGHT	The right-arrow key	F1 to F12	The function keys from F1 to F12
ESCAPE	The Esc key	0 to 9	The number keys from 0 to 9
TAB	The Tab key	A to Z	The letter keys from A to Z

Key Events Example 1

```
public class KeyEventDemo extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create a pane and set its properties
    Pane pane = new Pane();
    Text text = new Text(20, 20, "A");

    pane.getChildren().add(text);
    text.setOnKeyPressed(e -> {
      switch (e.getCode()) {
        case DOWN: text.setY(text.getY() + 10); break;
        case UP:  text.setY(text.getY() - 10); break;
        case LEFT: text.setX(text.getX() - 10); break;
        case RIGHT: text.setX(text.getX() + 10); break;
      default:
        if (Character.isLetterOrDigit(e.getText().charAt(0)))
          text.setText(e.getText());
  }
});
```

Key Events Example 1

- In a *switch* statement for an **enum** type value, the *cases* are for the *enum* constants. The constants are unqualified.
- For example, using **KeyCode.DOWN** in the *case* clause will be wrong and produce an error.
- Only a focused node can receive **KeyEvent**. Invoking **requestFocus()** on **text** enables **text** to receive key input. This method must be invoked after the stage is displayed.

Key Events Example 2

```
public class ControlCircleWithMouseAndKey extends Application {
  private CirclePane circlePane = new CirclePane();

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Hold two buttons in an HBox
    HBox hBox = new HBox();
    hBox.setSpacing(10);
    hBox.setAlignment(Pos.CENTER);
    Button btEnlarge = new Button("Enlarge");
    Button btShrink = new Button("Shrink");
    hBox.getChildren().add(btEnlarge);
    hBox.getChildren().add(btShrink);

    // Create and register the handler
    btEnlarge.setOnAction(e -> circlePane.enlarge());
    btShrink.setOnAction(e -> circlePane.shrink());

    circlePane.setOnMouseClicked(e -> {
      if (e.getButton() == MouseButton.PRIMARY) {
        circlePane.enlarge();
      }
      else if (e.getButton() == MouseButton.SECONDARY) {
        circlePane.shrink();
      }
    });
```

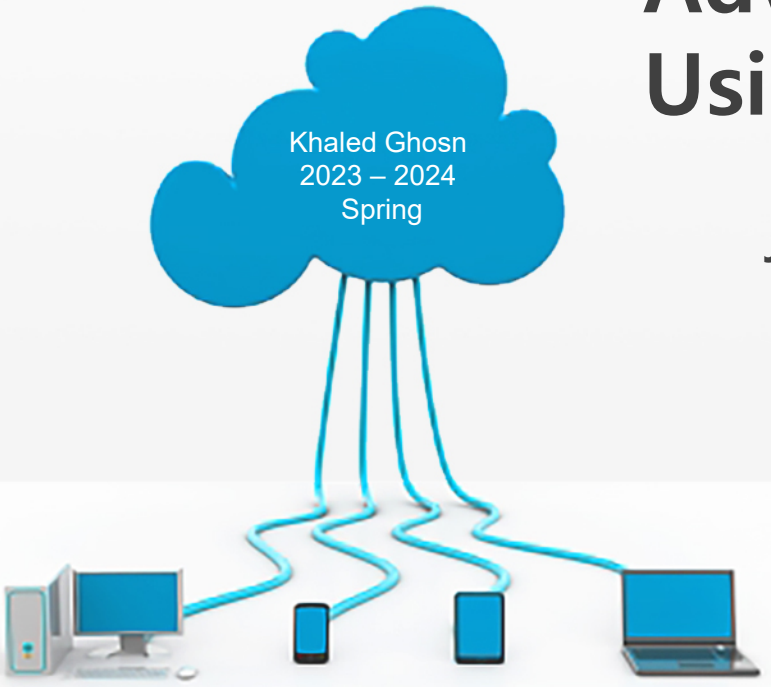
Key Events Example 2

```
    circlePane.setOnKeyPressed(e -> {
      if (e.getCode() == KeyCode.U) {
        circlePane.enlarge();
      }
      else if (e.getCode() == KeyCode.D) {
        circlePane.shrink();
      }
    });

    BorderPane borderPane = new BorderPane();
    borderPane.setCenter(circlePane);
    borderPane.setBottom(hBox);
    BorderPane.setAlignment(hBox, Pos.CENTER);

    // Create a scene and place it in the stage
    Scene scene = new Scene(borderPane, 200, 150);
    primaryStage.setTitle("ControlCircle"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    circlePane.requestFocus(); // Request focus on circlePane
  }
}
```

ARTS, SCIENCES & TECHNOLOGY
UNIVERSITY IN LEBANON
AUL

CCE 417

Advanced Programming Using Java



JavaFx Animation

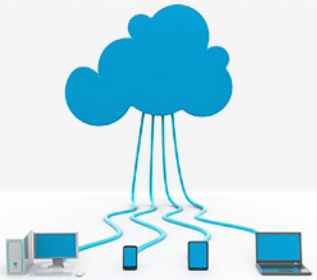
JavaFx Animation

Animation Basics

In general, the animation can be defined as the **transition** which creates the myth of motion for an object.

It is the set of transformations applied on an object over the specified **duration** **sequentially** so that the object can be shown as it is in motion.

This can be done by the rapid display of frames.



Animation in JavaFX can be divided into:

- ✓ **Transitions**
- ✓ **Timeline animation**

JavaFx Animation

Animation Basics

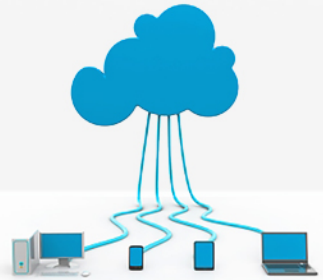
The JavaFX animation support enables you to animate JavaFX shapes and controls, e.g. moving or rotating them.

JavaFX provides easy to use animation API (javafx.animation package).

The package **javafx.animation** contains all the classes to apply the animations onto the nodes.

All the classes of this package extend the class

javafx.animation.Animation



JavaFx Animation

Animation Class

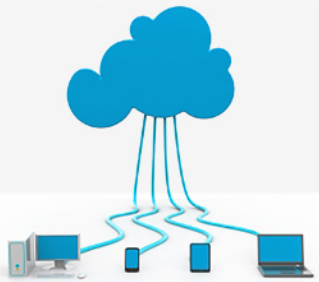
The class **Animation** provides the core functionality of all animations (used in the JavaFX runtime).

An animation can run in a loop by setting **cycleCount**.

To make an animation run back and forth while looping, set the **autoReverse** -flag.

Call **play ( )** or **playFromStart ( )** to play an Animation.

An Animation can be paused by calling **pause( )**, and the next **play( )** call will resume the Animation from where it was paused.



JavaFx Animation

Animation Class

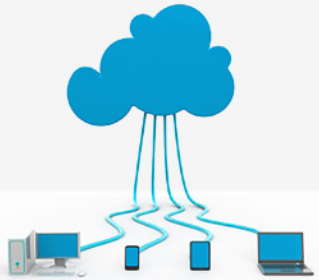
The Animation progresses in the direction and speed specified by **rate**, and stops when its **duration** is elapsed.

Inverting the value of **rate** toggles the play direction.

An Animation with **indefinite** duration (a **cycleCount** of **INDEFINITE**) runs repeatedly until the **stop( )** method is explicitly called, which will stop the running Animation and reset its play head to the initial position.

Animation in JavaFX can be divided into:

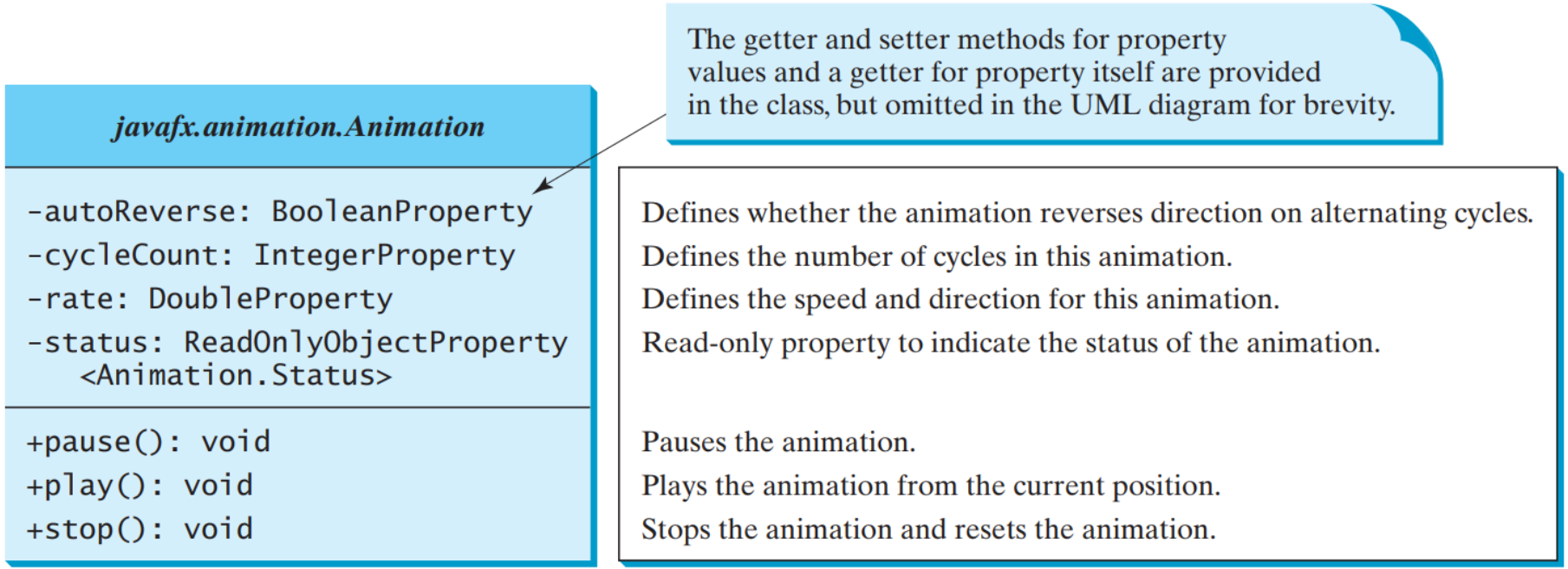
- ✓ **Transitions**
- ✓ **Timeline animation**



JavaFx Animation

Animation Class

The abstract **Animation** class is the root class for JavaFX animations



JavaFx Animation

Basic Transitions

Transition	Description
Fade Transition	creates a fade effect animation that spans its duration
Fill Transition	creates an animation, that changes the filling of a shape over a duration
Parallel Transition	plays a list of animations in parallel
Path Transition	creates a path animation that spans its PathTransition.duration
Pause Transition	executes an Animation.onFinished at the end of its PauseTransition.duration
Rotate Transition	creates a rotation animation that spans its duration
Scale Transition	creates a scale animation that spans its ScaleTransition.duration
Sequential Transition	plays a list of Animations in sequential order
Stroke Transition	creates an animation, that changes the stroke color of a shape over a duration
Translate Transition	creates a move/translate animation that spans its TranslateTransition.duration

Transitions

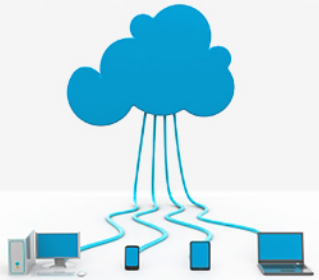
Path Transition

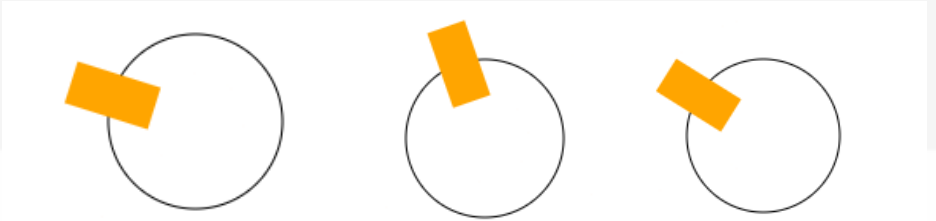
The **PathTransition** class animates the moves of a node along a path from one end to the other over a given time

The **Duration** class defines a duration of time.

The class defines constants **INDEFINTE**, **ONE**, **UNKNOWN**, and **ZERO** to represent an indefinite duration, 1 milliseconds, unknow, and 0 duration.

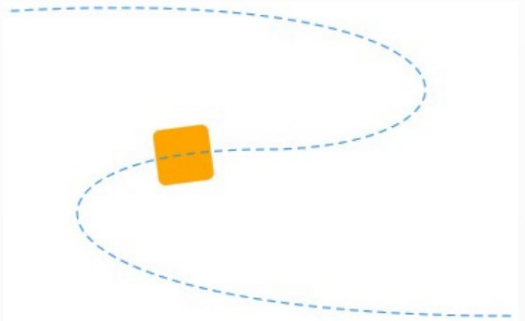
You can use new Duration(double millis) to create an instance of Duration





Transitions

Path Transition



javafx.animation.PathTransition

```
-duration: ObjectProperty<Duration>
-node: ObjectProperty<Node>
-orientation: ObjectProperty
   <PathTransition.OrientationType>
-path: ObjectType<Shape>
```

```
+PathTransition()
+PathTransition(duration: Duration,
   path: Shape)
+PathTransition(duration: Duration,
   path: Shape, node: Node)
```

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

The duration of this transition.

The target node of this transition.

The orientation of the node along the path.

The shape whose outline is used as a path to animate the node move.

Creates an empty `PathTransition`.

Creates a `PathTransition` with the specified duration and path.

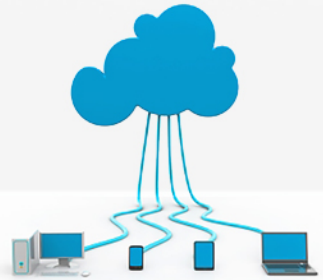
Creates a `PathTransition` with the specified duration, path, and node.

Transitions

Fade Transition

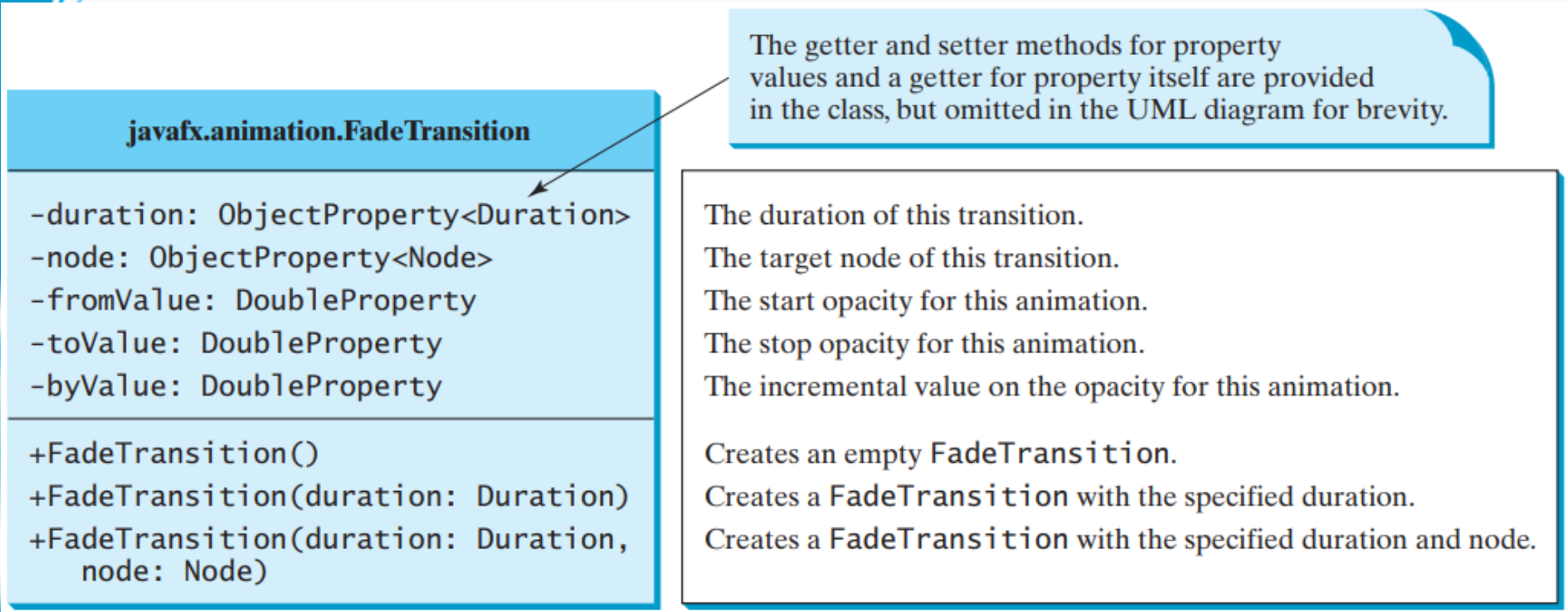
FadeTransition class animates the change of the opacity in a node over a given time.

FadeTransition creates a fade effect animation that spans its duration. This is done by updating the opacity variable of the node at regular interval.



Transitions

Fade Transition

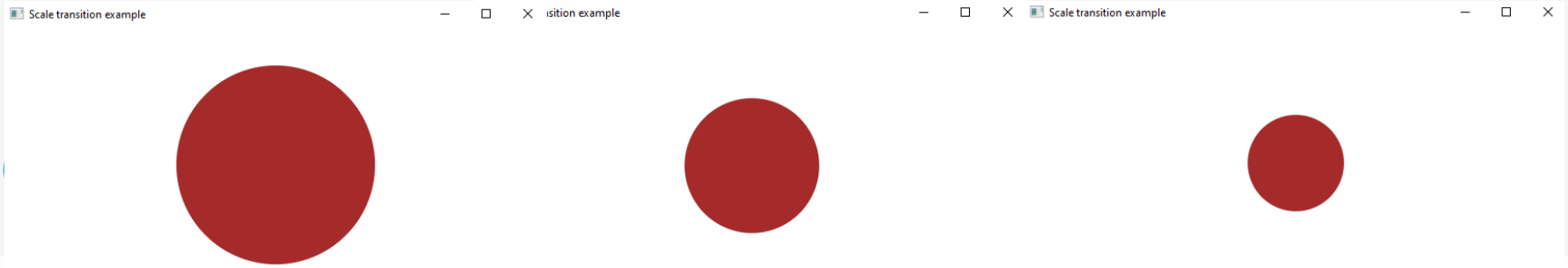


Transitions

Scale Transition

Scale transition is another JavaFX animation which can be used out of the box that allows to animate the scale / zoom of the given object

The object can be enlarged or minimized using this animation.

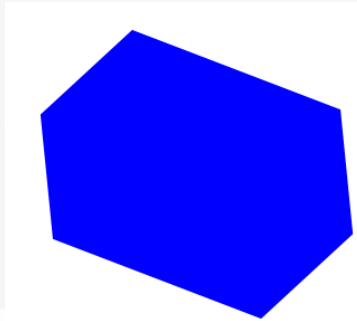


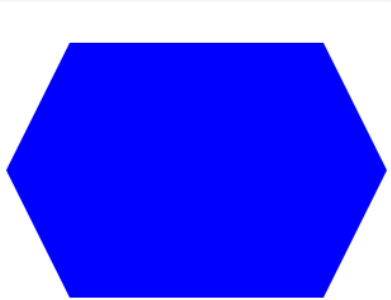
Transitions

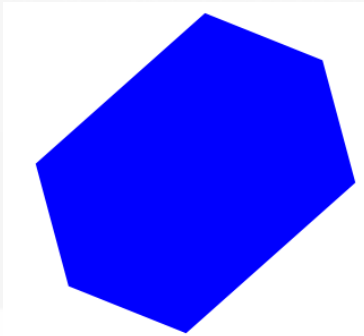
RotateTransition

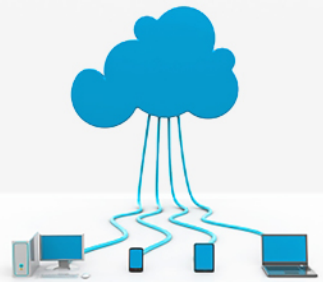
Rotate transition provides animation for rotating an object.

We can provide up to what angle the node should rotate by **toAngle**.
Using **byAngle** we can specify how much it should rotate from current angle of rotation.









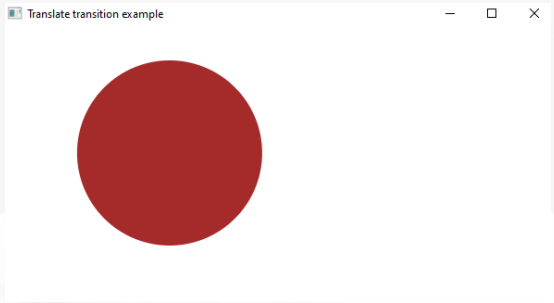
Transitions

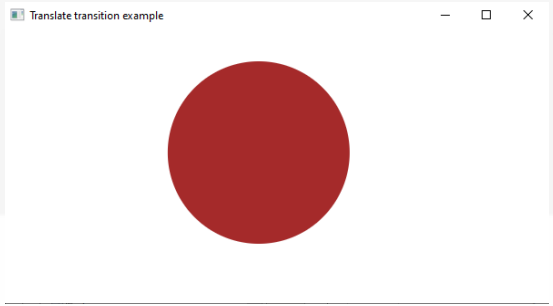
Translate Transition

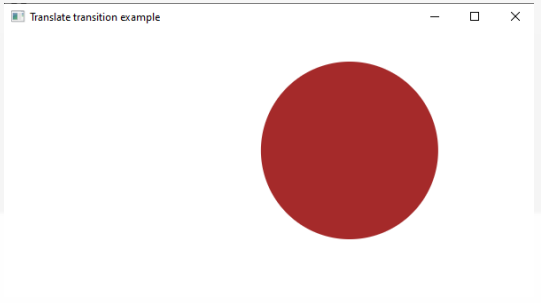
Translate transition allows to create movement animation from one point to another within a duration.

Using **TranslateTransition#setByX / TranslateTransition#setByY**, you can set how much it should move in x and y axis respectively.

It also possible to set precise destination by using **TranslateTransition#setToX / TranslateTransition#setToY**.





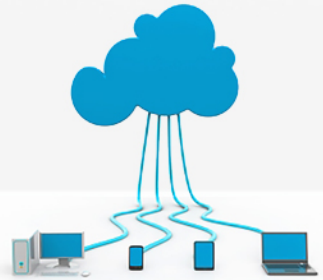


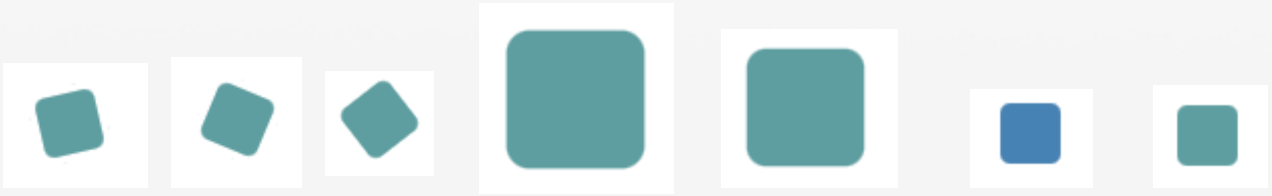
Transitions

Sequential Transition

A **sequential transition** executes several transitions one after another.

The following example shows the rotate, scale, and fill transitions applied to a rectangle.



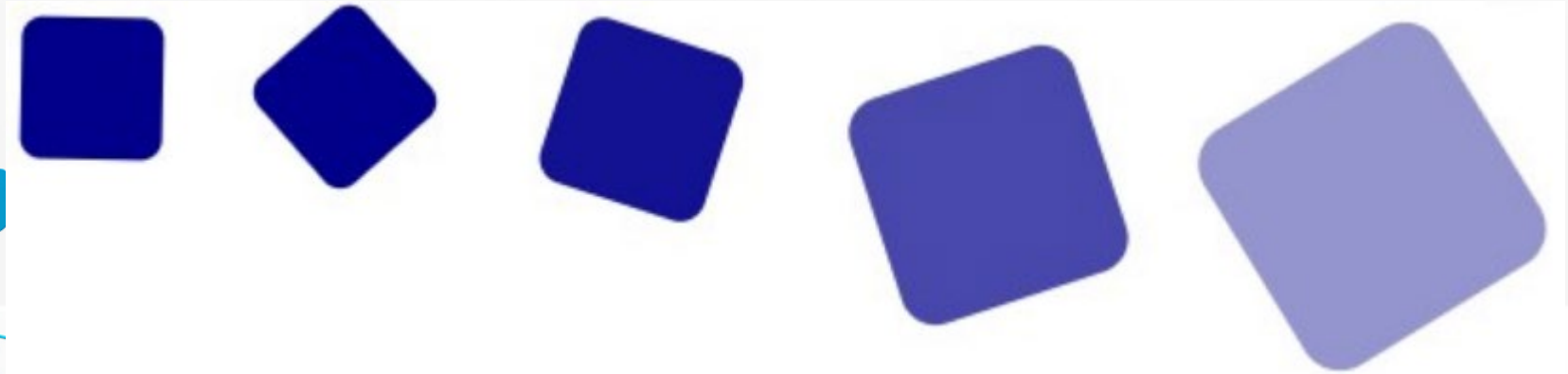


Transitions

Parallel Transition

A **parallel transition** executes several transitions simultaneously.

The following example shows the fade, translate, rotate, and scale transitions applied to a rectangle.



Timeline Animation

A **Timeline** can be used to define a free form animation of any Writable Value

The **Timeline** class can be used to program any animation using one or more **KeyFrames** which contain the properties of nodes that change. These properties are encapsulated in KeyValues.

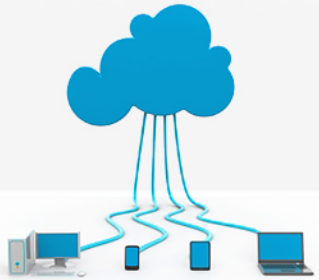
Each **KeyFrame** is executed sequentially at a specified time interval.

Timeline inherits from Animation.

You can construct a Timeline using the constructor

new Timeline (KeyFrame...keyframes)

A KeyFrame can be constructed using: *new KeyFrame (Duration duration, EventHandler<ActionEvent> onFinished)*



The handler **onFinished** is called when the duration for the key frame is elapsed.

Timeline Animation

Timeline inherits from Animation.

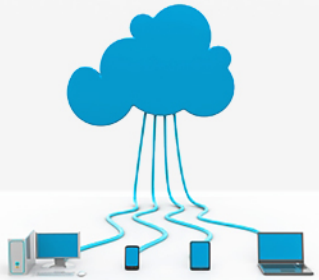
A **Timeline** can be used to define a free form animation of any Writable Value

The **Timeline** class can be used to program any animation using one or more **KeyFrames** which contain the properties of nodes that change. These properties are encapsulated in **KeyValues**.

KeyFrame defines target values at a specified point in time for a set of variables that are interpolated along a Timeline.

Each **KeyFrame** is executed sequentially at a specified time interval.

Timeline inherits from Animation.



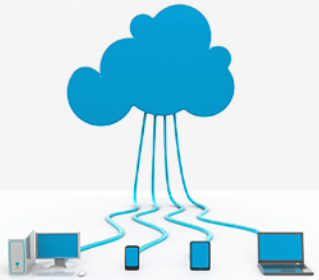
Timeline Animation

You can construct a Timeline using the constructor

new Timeline (KeyFrame...keyframes)

A **KeyFrame** can be constructed using: *new KeyFrame (**Duration** duration, EventHandler<ActionEvent> onFinished)*

The handler **onFinished** is called when the duration for the key frame is elapsed.

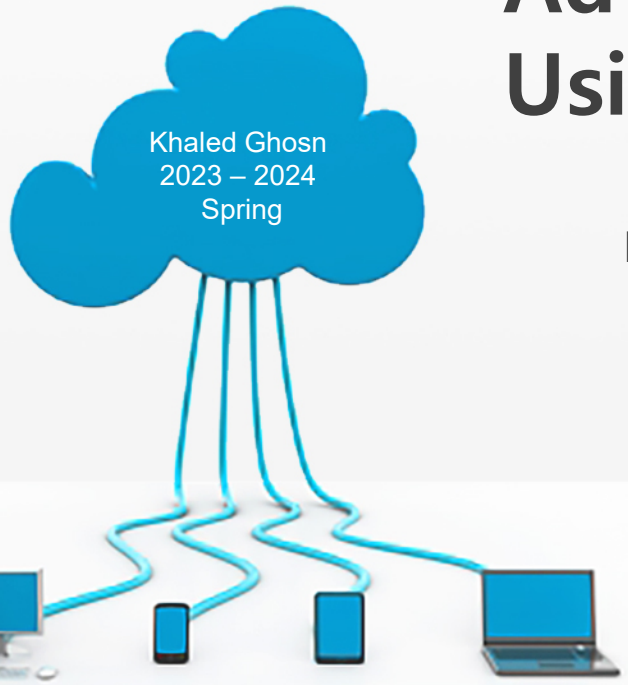


ARTS, SCIENCES & TECHNOLOGY UNIVERSITY IN LEBANON

AUL

CSC 320

Advanced Programming Using Java

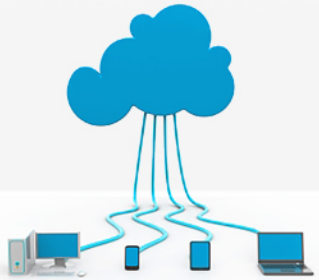


Khaled Ghosn
2023 – 2024
Spring

EXCEPTION HANDLING

Exception Handling

Exception handling enables a program to deal with exceptional situations and continue its normal execution.



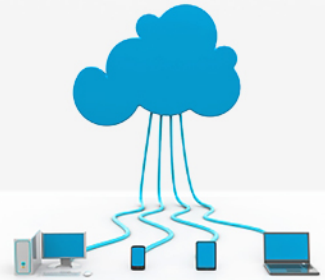
Exception Handling

Errors & Exception Types

a) **Compile Time Errors**
- E.g. syntax errors, static type mismatch, unresolved names, missing imports…
- Detected by the developer
- Program won't run before they're fixed

b) **Logical Errors**
- Happen at runtime without halting the program
- Undetectable by the compiler (otherwise program wouldn't run in the first place)
- Undetectable by the developer (otherwise he/she would have avoided it)
- E.g.
```
double tip = 10;
double subtotal = 25;
double total = subtotal * tip / 100;
```

c) **Runtime Errors**

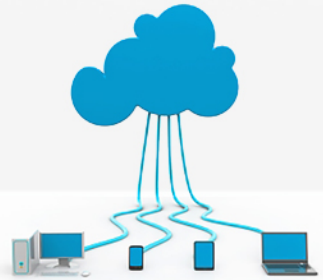


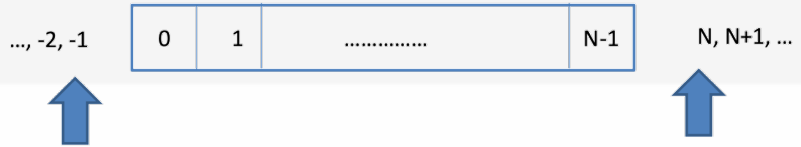
Exception Handling

Runtime Exceptions

c) Runtime Errors

- Happen at runtime (obviously!) causing the program to halt
- Undetectable by the compiler, but should be detected by the developer
- e.g.
  - division by 0 (ArithmeticException)
  - navigating inside a null object (NullPointerException)
  - Scanning type mismatch (InputMismatchException)
  - Traversing array outside its bounds (ArrayIndexOutOfBoundsException)
  - Passing illegal arguments to a method (IllegalArgumentException)



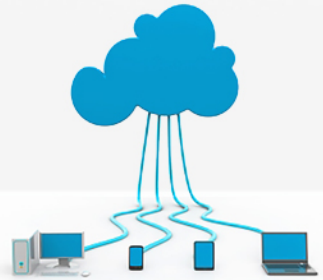


Exception Handling

The Exception Object

- In Java, runtime errors are thrown as exceptions.
- An exception is an object that represents an error or a condition that prevents execution from proceeding normally.
- We are responsible for handling exceptions so that the program can continue to run normally or terminate gracefully.

```
try {
  Code to run;
  A statement or a method that may throw an exception;
  More code to run;
}
catch (type ex) {
  Code to process the exception;
}
```

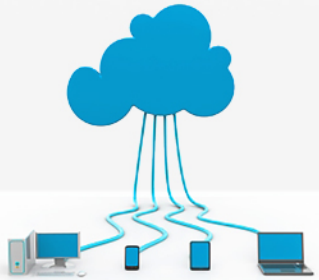


Exception Handling

Advantage of Using Exceptions Handling

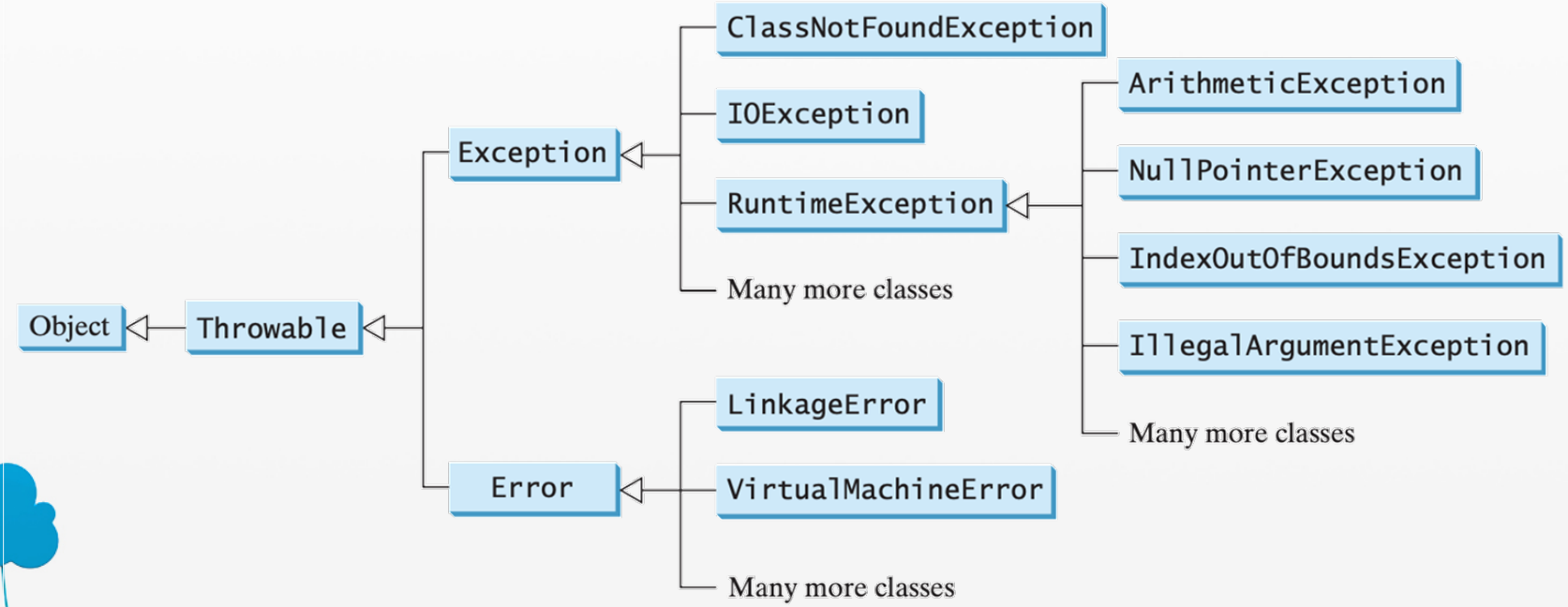
- Exception Handling enables a **method** to **throw** an exception to its caller, enabling the *caller* to *handle* the exception.

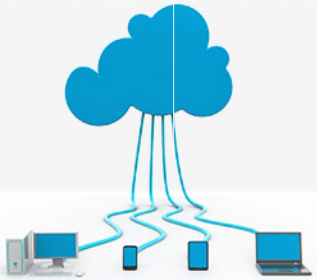
- So, the key benefit is separating the **detection of an error** (done in a *called* method) from the (done in the *calling* method). **handling of an error**.



Exception Handling

Exception Types

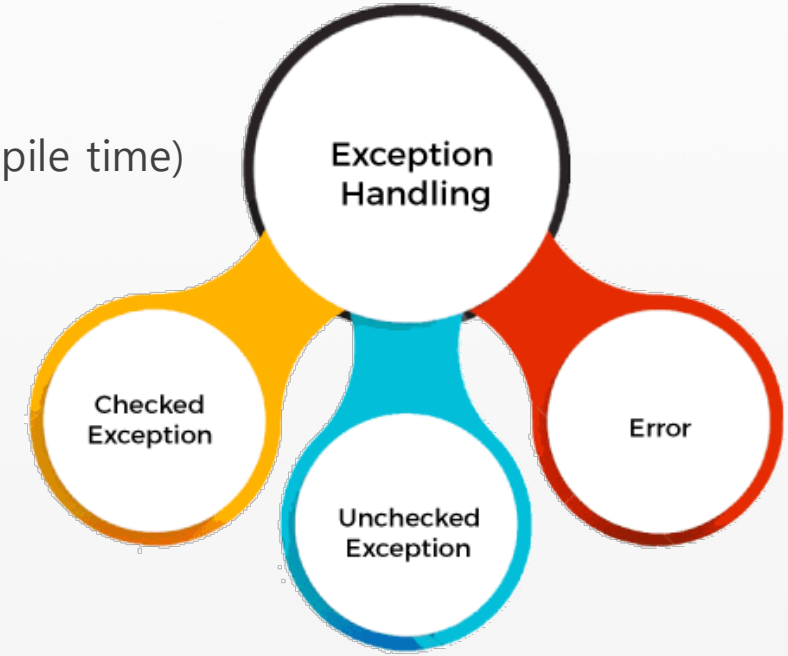


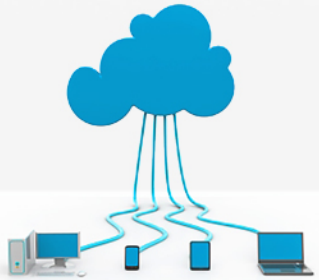


Exception Handling

Exception Types

- **Checked Exceptions** (at compile time)
- **Unchecked Exceptions**
- **Errors** (irrecoverable)





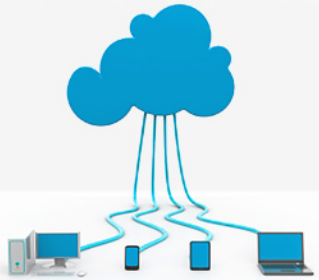
Exception Handling

Checked Exceptions vs. Unchecked Exceptions Types

- RuntimeException, Error and their subclasses are known as **unchecked** exceptions.

- All other exceptions are known as **checked** exceptions,

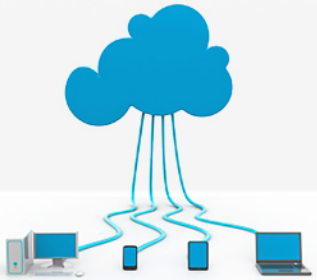
  Meaning that the compiler forces the programmer to check and deal with the exceptions either in a try-catch block or declare the exception in the method header.



Exception Handling

Unchecked Exceptions

- In most cases, unchecked exceptions reflect programming logic errors that are not recoverable, e.g.
  - a **NullPointerException** is thrown if you access an object through a reference variable before an object is assigned to it;
  - an **IndexOutOfBoundsException** is thrown if you access an element in an array outside the bounds of the array.
- These are the logic errors that should be corrected in the program.
- Unchecked exceptions can occur anywhere in the program.
- To avoid cumbersome overuse of try-catch blocks, Java does not mandate you to write code to catch unchecked exceptions.

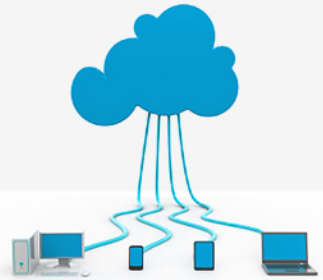


Exception Handling

Declaring, Throwing, and Catching Exceptions

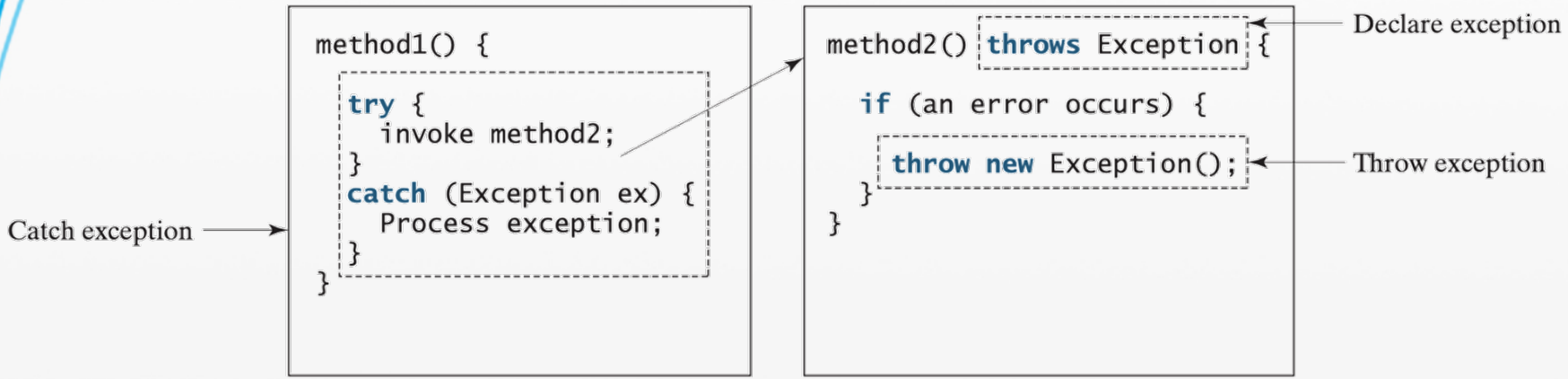
Java's exception-handling model is based on three operations:

1. **Declaring** an exception: throws
2. **Throwing** an exception: throw
3. **Catching** an exception

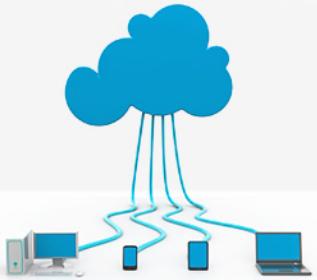


Exception Handling

Exception Handling Model



Exception handling in Java consists of declaring exceptions, throwing exceptions, and catching and processing exceptions.



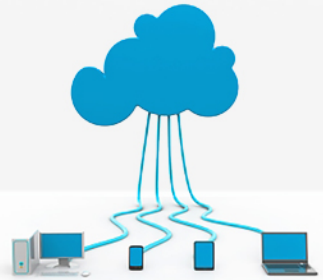
Exception Handling

Declaring an Exception

- Every method must state the types of checked exceptions it might throw.
- Use **throws** keyword in the method header to declare an exception:

  public void myMethod ( ) **throws** IOException
- Use separate commas to throw multiple exceptions

  public void myMethod ( ) **throws** Exception1, Exception2, ..., ExceptionN
- No need to try & catch inside the method when throwing.



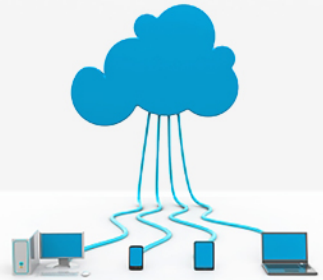
Exception Handling

Throwing an Exception

- A program that detects an error can create an instance of an appropriate exception type and throw it; e.g.
  - Exception ex = new Exception (“Something goes wrong");
    throw ex; // or you can write:
  - throw new Exception (" Something goes wrong");
- In general, each exception class in the Java API has at least two constructors:
  - a no-argument constructor, e.g.

    Exception ex = new Exception ( );
  - a constructor with a String argument that describes the exception. This argument is called the exception message, which can be obtained using **getMessage ( )** , e.g.

    ex.getMessage ( );



Exception Handling

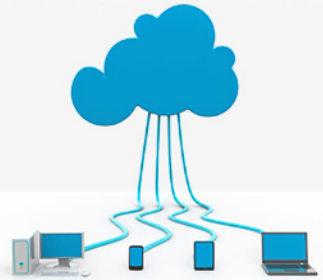
Throw and Throws in Java

Throw keyword:

- Is used to explicitly throw an exception from a method or any block of code.
- We can throw either checked or unchecked exception.
- The throw keyword is mainly used to throw custom exceptions.
- e.g. `throw new ArithmeticException ("/ by zero");`

Throws keyword:

- Is used in the signature of method to indicate that this method might throw one of the listed type exceptions.
- The caller to these methods has to handle the exception using a **try-catch block**.
- Syntax: type method_name (parameters) throws exception_list

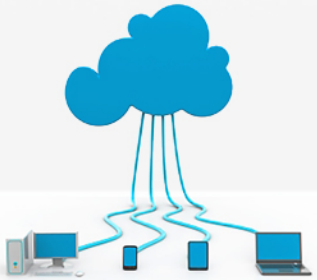


Exception Handling

Catching an Exception

When an exception is thrown, it must be caught.

```
try {
  statements; // Statements that may throw exceptions
}
catch (Exception1 exVar1) {
  handler for exception1;
}
catch (Exception2 exVar2) {
  handler for exception2;
}
...
catch (ExceptionN exVarN) {
  handler for exceptionN;
}
```



Handling several exceptions with same code:

```
catch (Exception1 | Exception2 | ... | Exceptionk ex) {
    // Same code for handling these exceptions
}
```

Exception Handling

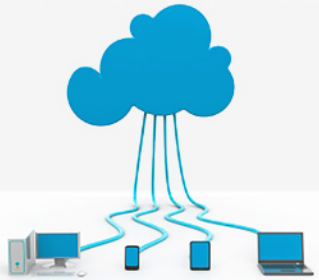
Catching an Exception

1. If no exceptions arise during the execution of the try block, the catch statements are skipped.

2. If one of the statements inside the try block throws an exception, the remaining statements are skipped. Then, the code will skip to the catch block which handles the exception that has occurred.

3. If no handler is found, the method is exited, the exception is passed to the calling method, and the same process is repeated.

4. If no handler is found in the chain of methods being invoked, the program terminates and prints an error message on the console.



Exception Handling

Order of catch blocks

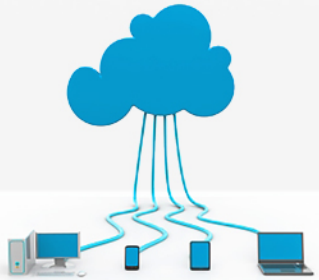
Catch the most specified first, then their parent, grandparents, and so on...

```
try {
  ...
}
catch (Exception ex) {
  ...
}
catch (RuntimeException ex) {
  ...
}
```

(a) Wrong order

```
try {
  ...
}
catch (RuntimeException ex) {
  ...
}
catch (Exception ex) {
  ...
}
```

(b) Correct order



Exception Handling

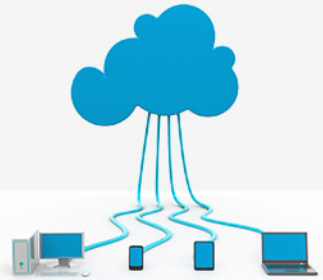
Order of catch blocks

- If the protected code can throw different exceptions which are not in the same inheritance tree, i.e. they don't have parent-child relationship, the catch blocks can be sorted any order.

- If the exceptions have parent-child relationship, the catch blocks must be sorted by the most specific exceptions first, then by the most general ones.

→

When you are handling multiple catch blocks, make sure that you are specifying exception sub classes first, then followed by exception super classes. Otherwise we will get compile time error.



Exception Handling

Catching an Exception

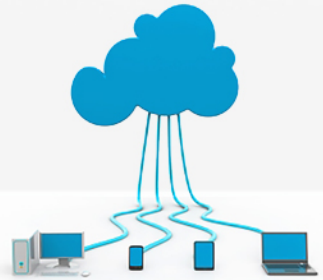
```
main method {
  ...
  try {
    ...
    invoke method1;
    statement1;
  }
  catch (Exception1 ex1) {
    Process ex1;
  }
  statement2;
}
```

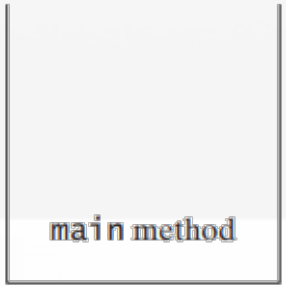
```
method1 {
  ...
  try {
    ...
    invoke method2;
    statement3;
  }
  catch (Exception2 ex2) {
    Process ex2;
  }
  statement4;
}
```

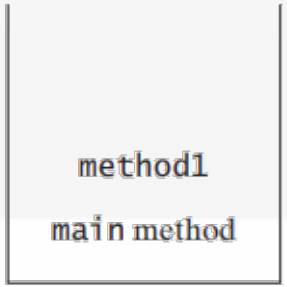
```
method2 {
  ...
  try {
    ...
    invoke method3;
    statement5;
  }
  catch (Exception3 ex3) {
    Process ex3;
  }
  statement6;
}
```

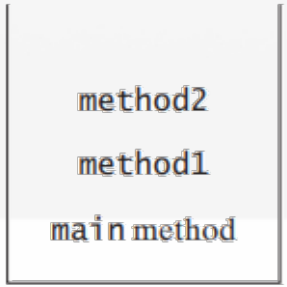
An exception is thrown in method3

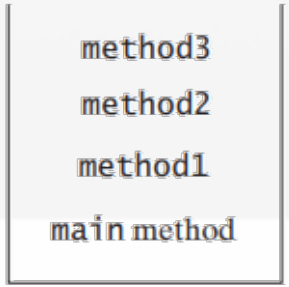
Call stack











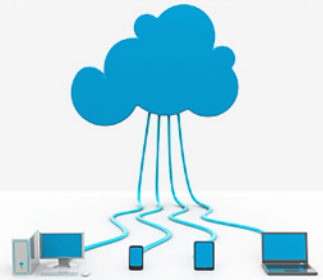
Exception Handling

The finally Clause

The finally clause is always executed regardless whether an exception occurred or not

(executed under all circumstances, regardless of whether an exception occurs in the try block or whether the exception is caught or not)

```
try {
    statements;
}
catch (TheException ex) {
    handling ex;
}
finally {
    finalStatements;
}
```



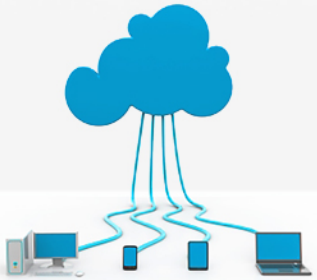
Exception Handling

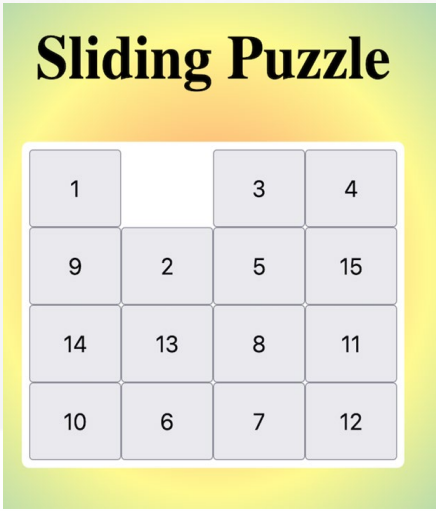
Creating Your Own Exception Class

You can define a custom exception class by extending the **java.lang.Exception** class.

E.g.

```
class IllegalMoveException extends
Exception {

    @Override

    getMessage()

    ...

}
```



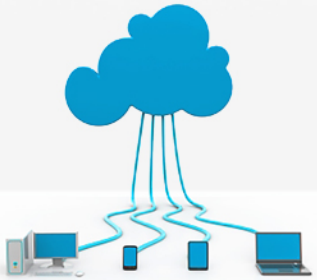


Exception Handling

Creating Your Own Exception Class

Then, in any method, declare it as:

```
public void move() throws IllegalMoveException {
…
    if(...) {
        throw new IllegalMoveException(...);
    }
}
```



Exception Handling

Getting Information from Exceptions

An exception object contains valuable information about the exception.

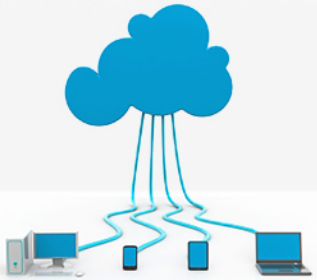
java.lang.Throwable	
`+getMessage(): String`	Returns the message that describes this exception object.
`+toString(): String`	Returns the concatenation of three strings: (1) the full name of the exception class; (2) `":"` (a colon and a space); (3) the `getMessage()` method.
`+printStackTrace(): void`	Prints the `Throwable` object and its call stack trace information on the console.
`+getStackTrace(): StackTraceElement[]`	Returns an array of stack trace elements representing the stack trace pertaining to this exception object.

Exception Handling

Rethrowing Exceptions

Java allows an exception handler to rethrow the exception if the handler cannot process the exception or simply wants to let its caller be notified of the exception.

```
try {
    statements;
    statements;
}
catch ( TheException ex ) {
   perform operations before exits;
   throw ex;
}
```



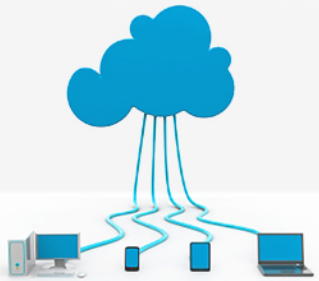
The statement **throw ex** rethrows the exception to the caller so that other handler in the caller get a chance to process the exception **ex**.

Exception Handling

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

statement2 throws an exception of type Exception2.

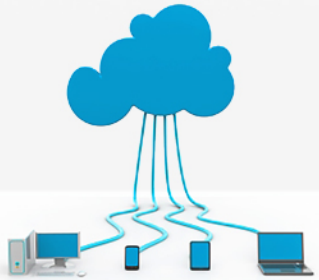


Exception Handling

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

Handling exception

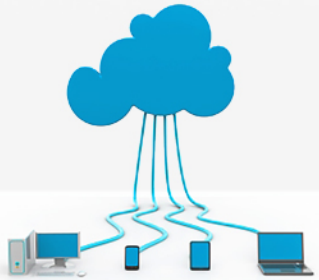


Exception Handling

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

Execute the final block

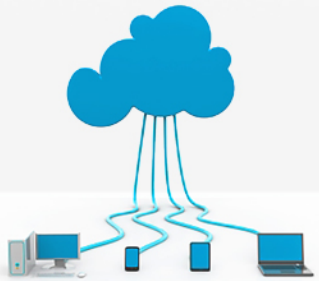


Exception Handling

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

Rethrow the exception and control is transferred to the caller

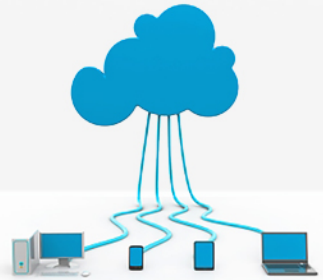


Exception Handling

Chained Exceptions

Sometimes, you may need to throw a new exception (with additional information) along with the original exception.

```
try {
  statements;
}
catch(TheException ex) {
  perform operations before exits;
  throw new Exception("Message", ex);;
}
```

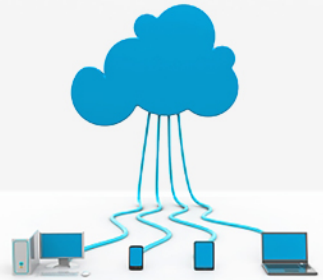


The exception ex is wrapped in a new exception, which is thrown and caught in the catch block in the caller method. The new exception will contain the original exception, in addition to its own parameters.

Exception Handling

Chained Exceptions

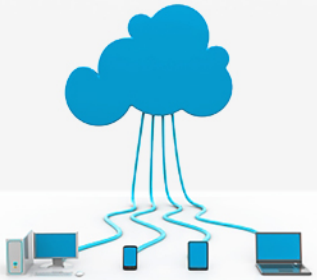
```
public class ChainedExceptionDemo {
  public static void main(String[] args) {
    try {
      method1();
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
  }

  public static void method1() throws Exception {
    try {
      method2();
    }
    catch (Exception ex) {
      throw new Exception("New info from method1", ex);
    }
  }

  public static void method2() throws Exception {
    throw new Exception("New info from method2");
  }
}
```



Exception Handling

Chained Exceptions

```
java.lang.Exception: New info from method1
  at ChainedExceptionDemo.method1(ChainedExceptionDemo.java:16)
  at ChainedExceptionDemo.main(ChainedExceptionDemo.java:4)
Caused by: java.lang.Exception: New info from method2
  at ChainedExceptionDemo.method2(ChainedExceptionDemo.java:21)
  at ChainedExceptionDemo.method1(ChainedExceptionDemo.java:13)
  ... 1 more
```



What would be the output if the body of the catch block in method1 is replaced by the following line?

```
throw new Exception("New info from method1");
```