VŠB TECHNICKÁ UNIVERZITA OSTRAVA

www.vsb.cz

Java

Jan Kožusznik, David Ježek
david.jezek@vsb.cz

Tel: 597 325 874
Room: EA406

Literature

- Sun.The Java™ Tutorials., http://download.oracle.com/javase/tutorial/
- Oracle Academy: Course „Java Foundations“
- Oracle Academy: Course „Java Programming“
- SCHILDT, Herbert, 2018. Java: A Beginner’s Guide, Seventh Edition. 8 edition. New York: McGraw-Hill Education. ISBN 978-1260440218 .

1th lecture - objectives

- Types, Operators, variables
- Object type, class structure
- Constructors
- Overloading

Motivation for Java

Motto:

„Write once, run anywhere“

Sun Microsystem

Features of Java technology

- Multiplatform and portable
- Object Oriented
- It has simple language – core is API
- Robust, Dynamic and Secure
- Multithreaded
- Support for distributed application
-

Data Types

- **Primitive types** – only values:
  - **int** is in [-2147483648, 2147483647]
  - **double** is in [4.9*10-324, 1.7976931348623157*10308]
  - **boolean** is in {false, true}
- **Object types** – reference to instance of class:
  - type from Java (more than 18000) – e.g. String
  - defined by user – e.g. Rectangle, Person

Primitive Types

- Similar to **C/C++ but**:
  - Types has exactly same size on every platforms
  - All numeric types are signed
  - boolean type is separate type and numeric types are not automatically converted in.
  - Type for strings (String) is object type
- **Integer** data types:
  - **byte** (8b), **short** (16b), **int** (32b), **long** (64b)
    - Their literals should contain '_' …. 10_000
    - long literals are defined with suffix l … 10l
- **Floating point** (Real) data type
  - **float** (32b), **double** (64b)
    - float literals are defined with suffix f … 3.151f
- **Textual primitive** type - **char** (16 b) – only single 16 bit Unicode character (0-65535)
- **Boolean** type – boolean (1b)

Operators

- Mainly for primitive types – exception is ‘+’ used for string concatenation and ‘[]’ used for arrays.
- Like C:
  - **unary**: +,-, ++,--
  - **binary**: …, modulo % also available for **double**
  - **assignment**: =, +=,-=, …
  - **relational**: ==, !=, <=, … operands are values of some numeric type (integer or real) result is value of boolean type.
  - **logical**: !, ||, && ,^ - operands and result **are always values of boolean type**
    - available also non lazy version |, & - both operands are always evaluated
  - **ternary**: <condition expression>?<value1>:<value2>
  - **bitwise**:
  - **cast**: () – automatic casting of value is allowed to a type that has bigger range(numeric primitive) or to parents (object)
- Construct expression with defined precedence.

Object Type

- An Object is an distinguishable entity that has:
  - **Identity**: a uniqueness which distinguishes it from all other objects
  - **Behavior**: services it provides to another objects
  - **State**: value of attributes held by an object
- A class is an abstraction of objects with similar implementation
  - Class is definition of set of similar objects
  - Every object is an instance of one class

Object is an instance of a class

- Memery is allocated, object is created and reference to the **instance** is stored into variable.

Instance of class Rectangle is created.

```
Rectangle rectangle1 = new Rectangle(); //an object creation
```

Identifiers - name variables, functions, classes, and objects - anything that programmers need to identify and use. Identifiers start with letter, underscore or dollar sign and they are case-sensitive. More about convention: http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html

Object responds to a message call

- Methods are called on object only by ‘.’ (not by ->)

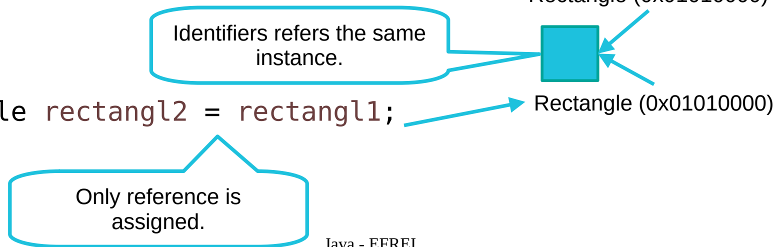
```
rectangle1.moveDown();
```

Message on the instance could be sent.

Reference vs. instance

- Another instance is created only by operation **new**.
- Reference to the same instance is passed during assignment.

```
Rectangle rectangl1 = new Rectangle();
rectangl1.moveDown();

//...
Rectangle rectangl2 = rectangl1;
```



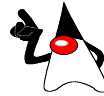
Comparing Variables (values)

- When you compare values by using **boolean** expressions, you need to understand the nuances of certain data types.
- Relational operators such as == are …
  - Great for comparing primitives
  - Terrible for comparing Strings (and other objects)

Comparing Primitives

- The value z is set to be the sum of x + y.
- When a boolean expression tests the equality between z and the sum of x + y, the result is true.

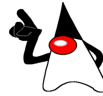
```java
int x = 3;
int y = 2;
int z = x + y;

boolean test = (z == x + y);
System.out.println(test); // true
```



Comparing Strings (true for objects)

- The value z is set to be the concatenation of x + y.
- When a boolean expression tests the equality between z and the concatenation of x + y, the result is **false**.

```
String x = "Ora";
String y = "cle";
String z = x + y;

boolean test = (z == x + y);
System.out.println(test); // false
```



Why Are There Contradictory Results?

- Primitives and objects are stored differently in memory.
  - *Strings* are given special treatment.
  - This is discussed later in the course.
- As a result ...
  - == compares the values of primitives.
  - == compares the objects' locations in memory.
- It's much more likely that you'll need to compare the content of Strings and not their locations in memory.

How Should You Compare Strings?

- You should almost never compare *Strings* using ==.
- Instead, compare Strings using the equals() method.
  - This method is part of the String class (part of every class).
  - It accepts one String argument, checks whether the contents of Strings are equal, and then returns a boolean.
  - There is also a similar method, equalsIgnoreCase()

```
String x = "Ora";
String y = "cle";
String z = x + y;

boolean test = z.equals(x + y);
System.out.println(test); // true
```

Variables

- Again similar to C/C++ (instance, local, static, methods arguments) **except**:
  - There is **no global variable** – every declaration should be placed inside class or their methods or other blocks
- default value depends on data type and variable type (local, instance, static) – **local variables need explicit definition of initial value**

Accessing Uninitialized Variables

- If variables aren't initialized, they take on a **default value**.
- **Not true for local variables!!!!!**
- Java provides the following default values:

Data Type	Default Value
boolean	false
int	0
double	0.0
String	**null**
Any Object type	**null**

Defining constants

- variable with modifier **final** – its value cannot be changed

```
private final int year;
```

- It is good practice to define variable as final when it is not changed in the future.
- Instance variable needs to be initialized in a constructor or by default value during declaration.

```
private final int year = 2024;
```

Null Object reference

- Variables of object type can have a **null** value.
- A **null** object points to an empty location in memory
- If an Object has another Object as a field (such as a String), its default value is **null**.
- What if a null object contains a field or method that needs to be accessed?
  - This causes the program to crash!(It is possible to handle it!)
  - The specific error is a **NullPointerException**.

```
public static void main(String[] args) {
  String test = null;
  System.out.println(test.length());
}
```

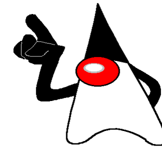
Java Classes in Source Code

Definition of class

- Every class have to be within own source file named “<class-name>.java” - following class Person is in file Person.**java**.
- Name of class should follow conventions -
- **Name should be noun, in mixed case with the first letter of each internal word capitalized.**
- All class definitions are inside class block ({})
- Visibility modifiers(**public**, **private**, **protected**) should be placed before every defined element.

Structure of a Class

```
public class Person {
  private LocalDate birthDay;
  private int actualIq;

  public Person() {
    this(LocalDate.now());
  }
  public Person(LocalDate aBirthDay) {
    this(aBirthDay, 110);
  }
  public Person(LocalDate aBirthDay, int actualIq) {
    birthDay = aBirthDay;
    this.actualIq = actualIq;
  }
  public void run(int maxSpeed) {
    // process of running
  }
  private int getActualAge() {
    int result;
    result = Period.between(birthDay, LocalDate.now()).getYears();
    return result;
  }
}
```

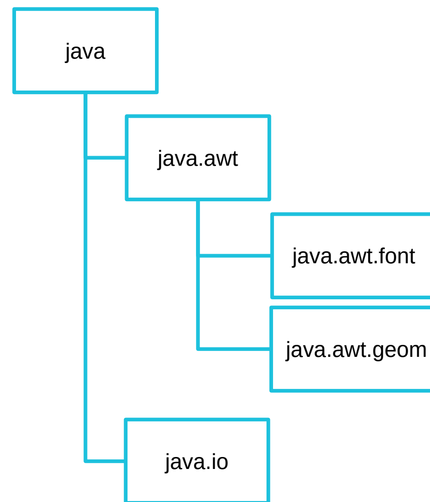


Class and Constructors

- Class constructor is always called when object is created (using keyword **new**)
- If constructor is not deffined, Java automatically create default empty constructor without parameters.

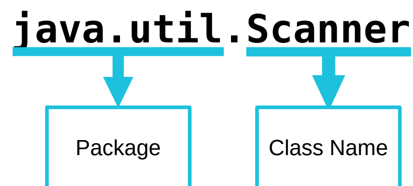
How Are the Packages Organized?

- The vast collection of classes are organized Into a tree-like hierarchy, which allows packages to be divided into subpackages, like this:



Using a Class from a Package

- To use a class from a package in your program, you need to specify its fully qualified name.
- For example, to use the **Scanner** class to read a keyboard input the fully qualified name for the **Scanner** class, which is defined in the **java.util** package is



Using the Full Qualified Class Name

```
public static void main(String[] args) {
  int num;
  java.util.Scanner keyboard = new java.util.Scanner(System.in);
  System.out.println("Enter a number");
  num = keyboard.nextInt();
  System.out.println("The entered number is " + num);
}
```

- As you can see, using the fully qualified name creates very long names for classes.
- Long names reduce the readability of the code and also make coding difficult.

Using the `import` Statement

- You can avoid the fully qualified class name by using the **import** statement.
- You place the **import** statement aboce your class definition. It looks like this:

```
import package.ClassName;
```

- Example:

```
import java.util.Scanner;
public class Numbers {
  public static void main(String[] args) {
    int num;
    Scanner keyboard = new Scanner(System.in);
```

Accessing All classes from the `java.util` Package

- As you access more classes from the **java.util** package in your program, the number of import statements also increases.
- To avoid this, you can import all classes from the **java.util** package by using the ***** wildcard character in the import statement, like this:

```
import java.util.*;
//import all class names
//from package java.util
```

Visibility modifiers

- They are used with:
  - variables – both instance and class
  - methods
  - ...and also classes!

access	public	protected	<none>	private
The same class	YES	YES	YES	YES
The same package	YES	YES	YES	NO
successors	YES	YES	NO	NO
anywhere	YES	NO	NO	NO

```
public class MyClass {

  private int a1;

  String a2;

  public double a3;

  public void method1() {
  }

  void method2() {
  }

  private void method3() {
  }
}
```

Overloading Constructors

- You can write more than one constructor in a class.
  - This is known as **overloading** a constructor.
  - A class may have an unlimited number of constructors.
- Each overloaded constructor is named the same.
- But they differ in any of the following ways:
  - Number of parameters.
  - Types of parameters.
  - Ordering of parameters.

Recognizing Redundancy in Constructors

- Very similar code is repeated in these constructors.
- It's possible to minimize this redundancy.

```
public Rectangle(int x, int y, int width, int height) {
  this.x = x;
  this.y = y;
  this.width = width;
  this.height = height;
  this.color = Color.RED;
}

public Rectangle(int x, int y, int width, int height, Color color) {
  this.x = x;
  this.y = y;
  this.width = width;
  this.height = height;
  this.color = color;
}
```

Constructors Can Call Other Constructors

- By using the **this** keyword, one constructor may call another.

```
public Rectangle(int x, int y, int width, int height) {
  this(x, y, width, height, Color.RED);
}

public Rectangle(int x, int y, int width, int height,
Color color) {
  this.x = x;
  this.y = y;
  this.width = width;
  this.height = height;
  this.color = color;
}
```

Overloading Methods

- Any method can be overloaded, including ...
  - Constructors
  - Methods that model object behaviors
  - Methods that perform calculations
- All versions of an overloaded method are named the same.
- But differ in any of the following ways(in a signature of the method):
  - **Number** of parameters
  - **Types** of parameters
  - **Ordering** of parameters
- Which version of overloaded methods is chosen **during compilation – important when we use object types and inheritance.**

Methods Can Call Other Methods in the Same Class

- In this example, one method returns a value to the other.

```
public class Calculator {
  public double calcY(double m, double x) {
    return calcY(m, x, 0);
  }

  public double calcY(double m, double x, double b) {
    return m * x + b;
  }
}
```

2nd lecture

- Basic OOP
  - Interface
  - Inheritance
  - Method overriding
- Scanner class
- Nested (Internal) class

Interface

- An interface is a Java construct that helps define the roles that an object can assume – it allows treat with objects of different classes uniformly
- It is implemented by a class or extended by another interface.
- An interface looks like a class with abstract methods (no implementation), but we cannot create an instance of it.
- Interfaces often define collections of related methods without implementations.
- All public methods in a Java interface are abstract (or default using another methods in the interface).

Why Use Interface

- When implementing a class from an interface we force it to implement all of the abstract methods.
- The interface forces separation of what a class can do, to how it actually does it.
- So a programmer can change how something is done at any point, without changing the function of the class.
- This facilitates the idea of polymorphism as the methods described in the interface will be implemented by all classes that implement the interface.

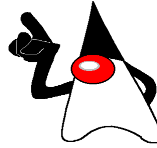
Interface properties

- An interface:
  - Can declare public constants.
  - Define methods without implementation, default method, private methods or static method.
  - Can only refer to its constants and defined methods or other accessible methods (static or methods of objects passed as parameter).
  - Can be used with the instanceof operator.
- A class
  - can implement more then one interface
- An interface method
  - Each method is public even when you forget to declare it as public – private methods are exception.
  - Is implicitly abstract but you can also use the abstract keyword.
  - Each variable is public final static – even without modifier.

Declaring Interface

- To declare a class as an interface you must replace the keyword class with the keyword interface.
- This will declare your interface and force all methods to be abstract and make the default access modifier public.

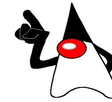
```
public interface Paintable {
  int DEFAULT_SIZE = 10;
  void paint(MyGraphics d);
}
```



Default (Java 8) and private (Java 9) methods

- These methods can not deal with inner structure
- Help remove redundancy in code and extend existing interface

```
public interface Movable {

  void setPosition(int x, int y);

  int getX();

  int getY();

  default void moveRight() {
    move(10, 0);
  }

  private void move(int dx, int dy) {
    setPosition(getX() + dx, getY() + dy);
  }
}
```



Interface Implementation

```
public class Rectangle implements Paintable {
// ...
  @Override
  public void paint(MyGraphics d) {
    // ...
  }
```

Class declares implementation of specified interface.

It has to implement every method of specified interface.

Multiple Interface implementation

- Every class could implement more then one interface.

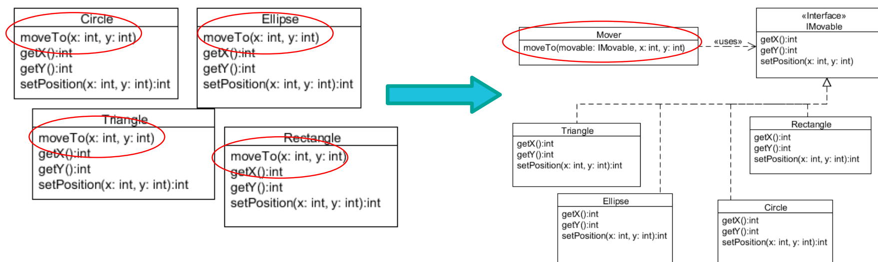
```
public class Rectangle implements Paintable, Clear {
// ...
```

- When are implemented two or more interfaces with same default methods then these methods should be overridden. It could call one of the existing implementations.

```
public class MyClass implements Movable, Pickable {
  @Override
  public void moveRight() {
    Movable.super.moveRight();
  }
```

Design pattern Template method

- Common logic is placed externally of class.
- Class is accessed through defined interface.



Design pattern Template method in source code

```
public class Mover {
  private static final long SLEEP_TIME_IN_MS = 500;
  private static final double SPEED = 10;

  public void move(IMovable object, int toRight, int toDown) {
    double distance = Math.sqrt(toRight * toRight + toDown * toDown);
    int STEPS = (int) (distance / SPEED);
    double dx = (toRight + 0.4) / STEPS;
    double dy = (toDown + 0.4) / STEPS;
    int xPos = object.getX();
    int yPos = object.getY();
    double x = xPos + 0.4;
    double y = yPos + 0.4;

    for (int i = STEPS; i > 0; i--) {
      x = x + dx;
      y = y + dy;
      object.setPosition((int) x, (int) y);
      Utils.sleep(SLEEP_TIME_IN_MS);
    }
  }
}
```

Common behavior is in a separate class

The algorithm – behavior will be applicable on any object that implements a specific interface

```
public interface IMovable {
    int getX();

    int getY();

    void setPosition(int x, int y);
}
```

Interface extension

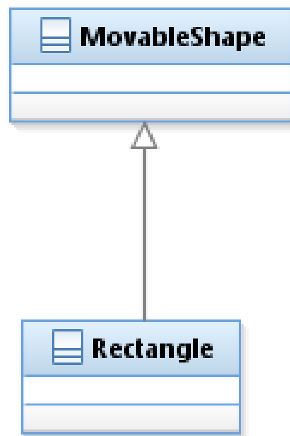
- If class implement the interface **IMovable** than it must also implement interface **IPaintable**.

```
public interface IMovable extends IPaintable {
```

Interface extends multiple interfaces

- Interface can extend from multiple interfaces.
- When are extended two or more interfaces with same default methods then these methods should be overridden as default - it could call one of the existing implementations (similar to implementation) – or leave them as abstract.

Class extension - inheritance

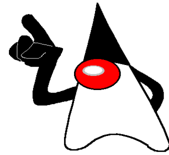


- **`Rectangle`** is specialization of **`MovableShape`**.
- **`MovableShape`** is generalization of **`Rectangle`**.
- **`Rectangle`** is subclass (successor) of **`MovableShape`**.
- **`MovableShape`** is superclass (predecesor) of **`Rectangle`**.

Inheritance in Java

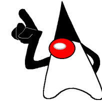
- Class could inherit only from one another class in Java.
- If a superclass is not specified than class inherits from class **Object**.

```
public class Rectangle extends MovableShape {
```



Inheritance of constructors

```
class ParentClassType {
  private int parentValue;

  public ParentClassType() {
    parentValue = 10;
  }

  public ParentClassType(int value) {
    parentValue = value;
    System.out.println(
      "Parent constructor called");
  }
}


public class ClassType
  extends ParentClassType {

}
```



- Constructors are not inherited in successors.

  ~~`new ClassType(10);`~~

- Only default non-parametric constructor is available in this case.

  `new ClassType();`

Implicit calling of predecessor constructor

```
class ParentClassType {
  private int parentValue;
  public ParentClassType() {
    System.out.println("Parent constructor called ");
  }
  public ParentClassType(
    int value) {
    System.out.println("Parent constructor called " + "with value " + value);
    parentValue = value;
  }
}
```

```
public class ClassType extends ParentClassType {
  private int value;
  public ClassType() {
    System.out.println("Child constructor called ");
    value = 0;
  }
  public ClassType(int value) {
    System.out.println("Child constructor called ");
    this.value = value;
  }
}
```

- Both constructors

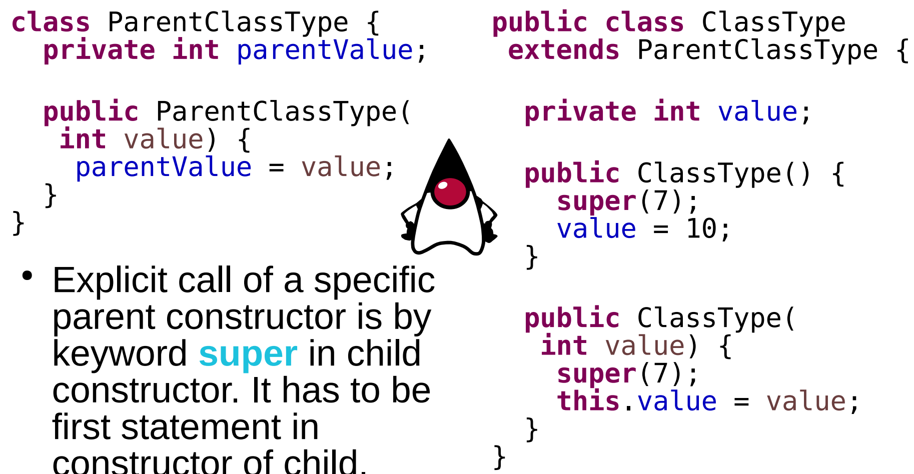
```
new ClassType();
new ClassType(10);
```

- Product output:

```
Parent constructor called
Child constructor called
```

- Non-parametric constructor of predecessor (ParentClassType) is implicitly called at the beginning of object construction

Explicit calling of predecessor constructors



```
class ParentClassType {
  private int parentValue;

  public ParentClassType(
   int value) {
    parentValue = value;
  }
}
```

- Explicit call of a specific parent constructor is by keyword **super** in child constructor. It has to be first statement in constructor of child.

```
public class ClassType
 extends ParentClassType {

  private int value;

  public ClassType() {
    super(7);
    value = 10;
  }

  public ClassType(
   int value) {
    super(7);
    this.value = value;
  }
}
```

Instance building

```
class ParentClassType {
  private int parentValue;

  public ParentClassType() {
  }

  public ParentClassType(int value) {
    parentValue = value;
  }
}
```

```
public class ClassType
 extends ParentClassType {
  private int value;

  public ClassType() {
    this(10);
  }

  public ClassType(int value) {
    this(value, 7);
  }

  public ClassType(int value,
   int parentValue) {
    super(parentValue);
    this.value = value;
  }

  public ClassType(String anotherValue) {
    System.out.println(
      "Child constructor called with value "
      + anotherValue);
  }
}
```

- If **this(…)** or **super()** are used, then it should be first statement in constructor. Otherwise nonparametric constructor of predecessor is called.

Disadvantage of inheritance

- It breaks encapsulation principle
- it is necessary to know details of implementation in super-class.
- Composition is preferred.

Class Lifecycle - Full

```
public class ClazzLifeCycle {

  static {
    System.out.println("Static Initializer 1");
    statField4 = 5;
  }

  private static final int statField1 = statMethod1();
  private static int statField2 = statMethod1();
  private static int statField3 = 5;
  private static final int statField4;

  static {
    System.out.println("Static Initializer 2");
  }

  private static int statMethod1() {
    System.out.println("Static method 1");
    return 1;
  }

  {
    System.out.println("instance initializer block 1");
    field5 = 5;
  }

  private final int field1 = statMethod1();
  private final int field2 = method1();
  private int field3 = method1();
  private final int field4 = 4;
  private final int field5;
  private final int field6;

  {
    System.out.println("instance initializer block 2");
  }

  public ClazzLifeCycle() {
    field6 = 6;
    System.out.println("Constructor");
  }

  private int method1() {
    System.out.println("Method 1");
    return 2;
  }

  public static void main(String[] args) {
    new ClazzLifeCycle();
  }

}
```


https://blogs.oracle.com/javamagazine/post/java-instance-initializer-block

Method overriding

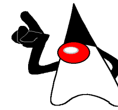
- Method redefined in successor are marked with **annotation** *`@Override`*.
- Overriding method should have **same signature** and compatible return type:
- same in case of a primitive type
- same or a subtype in case of an object type

```
class ParentClassType {
  public void methodA() {
    // some implementation
    // of methodA
  }
  public void methodB() {
  }
}

public class ClassType
 extends ParentClassType {

  @Override
  public void methodA() {
    // statements before
    super.methodA();
    // statements after
  }
  public void methodC() {
  }
}
```

Casting object variables, operator instanceof

- Variable of object type is implicitly casted to a supertype – predecessor or type of interface implemented by the given type.
- Variable of a object type could be explicitly casted to a its subtype:
  - An interface implemented by object
  - A class that is a super class or a class of the object.
- Casting to a unfit type fails during runtime – *`ClassCastException`* is thrown.
- Binary operator **instanceof** is used for testing whether object is given type

```
ParentClassType val = new ClassType();
ClassType val2 = (ClassType)val;
```



Nested (Internal) class

- Global – could be qualified with name/instance of outer class.
  - class – static internal class
  - instance – inner classes.
- Local – defined in block of code
  - Named,
  - Anonymous.

Static internal class

- A static internal class can only access static attributes of its containing (external)class.
- Internal class that does not need an object of the enclosing class to exist.
- Useful class for hiding layout details
- In another class, one can instantiate an internal class object, provided you use the visibility qualifier which is the name of the enclosing (external) class in order to access the internal class.
- The name of the internal class is **ExternalClass.InternalClass**

Static internal class

```
public interface IMovable {

  static public class MAdapter implements IMovable {
    // class implementation
  }

  public static void main(String[] args) {
    IMovable.MAdapter adapte =
      new IMovable.MAdapter();
  }
}
```

- Class nested type is qualified with name of outer type – if is needed.

Inner classes (non-static internal class)

- An internal Class instance needs an instance of the enclosing (external) class to exist.
- The internal class can access the fields of the object.
- During construction, an internal class must be constructed by an object of the enclosing class.
- It is forbidden to declare static members inside an inner-class, but it is possible to declare it at the level of its enclosing class.

Inner classes (non-static internal class)

```
public class OutterClass {
  private int outerVal;

  public class InnerClass {
    private int innerVal;

    public void setVal(int val) {
      // accessing feature of outer class
      OutterClass.this.outerVal = val;

      // accessing feature
      // of current class
      this.innerVal = val + 1;
    }
  }

  public InnerClass getInstance() {
    return new InnerClass();
  }
}
```

```
public static void main(
  String[] args) {
  OutterClass val =
   new OutterClass();
  OutterClass.InnerClass nextVal
   = val.new InnerClass();
}
```

Instance of inner class contains reference to an instance of outer class.

Instance of inner class could be in an instance method of outer class or with instance of outer class qualification.

Local classes

```
public void moveDown() {
  class MThread extends Thread {
    @Override
    public void run() {
      Baloon.this.run();
    }
  }
  new MThread().start();
}
```

Name local class.

```
public void moveUp() {
  Runnable run = new Runnable() {
    @Override
    public void run() {
      Baloon.this.run();
    }
  };
  new Thread(run).start();
}
```

Anonymous local class - “block of code” object that can be evaluated when needed.

- When a class definition is local to a block, it may access only attributes and constants

Anonymous class

- An anonymous class is a class that does not have a name.
- Since an anonymous class has no name, it is therefore not possible to define a constructor for it.
- An anonymous class is instantiated immediately in its declaration according to a specific syntax:

```
new <class identifier>(<list of construction parameters>) {
  <body of the class>
}
```

- An anonymous class is useful when a class is needed for single use, it is defined and instantiated where it is to be used.
- This also applies to interfaces using the following syntax:

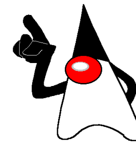
```
Interface c = new Interface() {
  // implementation of Interface methods
};
```

Supertype Class *Object*

- Root node in a hierarchy of all Java classes – supertype.
- Contains fundamental methods provided by all object:
  - toString
  - equals
  - hashCode
  - getClass
  - notify, notifyAll, wait
  - ...

Overriding of *equals* method

```
public class Fraction {
  final private int numerator;
  final private int denominator;

  public Fraction(int numerator, int denominator) {
    this.numerator = numerator;
    this.denominator = denominator;
  }

  @Override
  public boolean equals(Object obj) {
    if (obj instanceof Fraction) {
      Fraction other = (Fraction) obj;
      if (denominator == other.denominator
          && numerator == other.numerator)
        return true;
    }
    return false;
  }
}
```



Overriding of *toString* method

```
public class Fraction {
  final private int numerator;
  final private int denominator;

  public Fraction(int numerator, int denominator) {
    this.numerator = numerator;
    this.denominator = denominator;
  }
  @Override
  public String toString() {
    return numerator + "/" + denominator;
  }
}
```

Keyword *static*

- Used for variable – value is defined for given class instead of an object

```
public class Mover {
  private static final long SLEEP_TIME_IN_MILI_SECONDS = 100;
```

- Used for method – method is called on given class instead of an object (e.g. public)

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

Singleton classes

```
public class PrinterDriver {
// Static variable reference of driverInstance
// of type PrinterDriver
  private static PrinterDriver driverInstance
    = null;

  private String port;

// Private constructor restricted to
// this class itself, ensure only
// this class can create instance
  private PrinterDriver() {
    port = detectPort();
  }

// Static method to create instance
// of PrinterDriver class
  public static PrinterDriver getInstance() {
    if (driverInstance == null)
      driverInstance = new PrinterDriver();
    return driverInstance;
  }
}
```

```
public static void main(
 String[] args) {
// Instantiating PrinterDriver
// class with variable x
  PrinterDriver x =
   PrinterDriver.getInstance();

// Instantiating PrinterDriver
// class with variable y
  PrinterDriver y =
   PrinterDriver.getInstance();

  System.out.println(x == y); //true
}
```

Abstract Classes

- An abstract method has no body; it has a signature definition followed by a semicolon, e.g.

```
public abstract void method();
```

- Any class with an abstract method must be abstract – it needs keyword **abstract** before class.
- An abstract class cannot be instantiated.
- A subclass of an abstract class can be instantiated if it implements each of the abstract methods.
- This concept is defined a common predecesor for classes that share inner structure or implementation.

Virtual method

- All methods(functions) except **final**, **private** and **static** are virtual.
- Virtual machine (Java Hotspot) could make virtual method non-virtual or even inline during optimalization and conversion into a native code but prommer do not need take care about.

Complete meaning of a *final* modifier

- used with:
  - variable – its value cannot be changed

```
private final int year;
```

  - method – cannot be overridden

```
public final void someMethod() {/*implemntation*/}
```

- class – cannot be subclassed (be parent to another class)

```
public final class MyDate {
```

Exceptions

Handling of exceptional situations in Java

- Java uses system of exceptions as many other programming languages:
  - C++
  - C#
  - Python
  - PHP
  - Ruby

Technics to indicate that an error occurred

- diagnostic return value
  - Test the return value.
    - Attempt recovery on error.
    - Avoid program failure.
  - Ignore the return value.
    - Cannot be prevented.
    - Likely to lead to program failure.
- exception throwing (**preferred**)
  - Handle exception
  - Pass exception to another block of code where it should be handled

Exception-throwing principles

- Directly implemented in a language
- No ‘special’ return value needed.
- The normal flow-of-control is interrupted.
- Special recovery actions are supposed.
- Thrown exception cannot be ignored in the client object

How is an exception thrown

- An object representing an exception is constructed:

```
new ExceptionType("...")
```

- The keyword “throw” is used to throw the exception object:

```
throw new ExceptionType("a error message");
```

- If the method throws an exception outside, then it is specified in Javadoc documentation:

```
@throws ExceptionType reason description
```

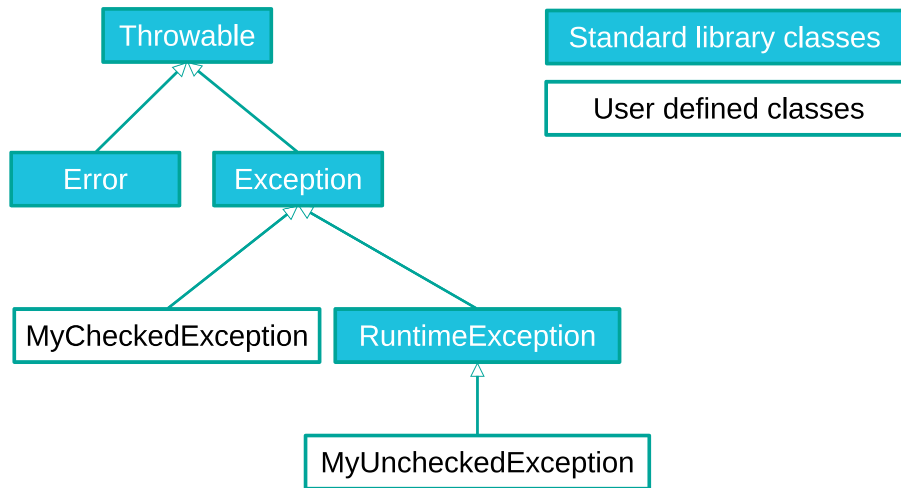
- An exception should be also thrown by Virtual machine internally (division by zero value)

Exceptions

- Its throwing indicates an exceptional situation,
- any instance of class inherited from **Throwable**
- important examples: **IllegalArgumetnException**, **IllegalStateException**, **NullPointerException**, **IndexOutOfBoundException**, **RuntimeException**

```
// immediate exception throwing
throw new IllegalArgumentException();

// exception could be throwed later
IllegalArgumentException e = new IllegalArgumentException();
throw e;
```

Overview of an exception class hierarchy



Main exception categories

- **Checked exceptions**
  - Subclass of **Exception**
  - Used for anticipated failures.
  - Where recovery may be possible.
  - Should be handled in a method when raises or the method should be marked
- **Unchecked exceptions**
  - Subclass of **RuntimeException**
  - Used for unanticipated failures.
  - can raise uncontrollably from a method

What exceptions does Java throw to us

- Error – InternalError, OutOfMemoryError, VirtualMachineError, StackOverflowError.
- Exception
  - unchecked- **RuntimeException** and its successor
    - ArithmeticException, IllegalArgumentException, UnsuportedOperationException, IllegalStateException,…
  - checked – other successor of **Exception**
    - ClassNotFoundException, DataFormatException, InterruptedException, IOException.

The effect of an exception

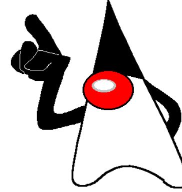
- The throwing method finishes exceptionally.
- The throwing method returns no value.
- Control does not return after the point of method call.
- A client must/may handle an exception.

Unchecked exceptions

- Compiler doesn’t check these exceptions
- It causes program termination if not handled.
- **NotSupportedException** is a typical example.

Example of an unchecked exception

```
  public void setDimension(int width, int height) {
    if ((width < 0) || (height < 0)) {
      throw new IllegalArgumentException(
          "Dimension should be greater or equal to 0:
width=" + width + ", height = " + height);
    }
    erase();
    this.width = width;
    this.height = height;
    paint();
  }
```

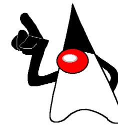


Exception handling

- Checked exceptions are perceived to be caught and eventually handled.
- The compiler ensures that their use is strictly controlled.
- Used carefully, failures may be recoverable.

Example of block with an exception

```
try {
  // critical section
} catch (IllegalArgumentException e) {
  // handle exception o type IllegalArgumentException
} catch (NullPointerException | IOException e) {
  // handle exception o type NullPointerException and
IOException
} catch (Throwable e) {
  // handle every exception – object o type Throwable
} finally {
  // this block is always performed when the critical section
is leaved
}
```



Throws clause

- Methods that can propagate a checked exception must be marked with throws clause:

```
/**
 *
 * @param fileName
 * @return
 * @throws IOException
 */
public String filterInputFile(String fileName) throws IOException {
  InputStream is = new FileInputStream(fileName);
  StringBuilder sb = new StringBuilder();
  int chars;
  byte[] buffer = new byte[1024];
  while (-1 != (chars = is.read(buffer))) {
    sb.append(new String(buffer, 0, chars));
  }
  is.close();
  return sb.toString();
}
```

Every checked exception should be handled in the method or the method specifies exception throwing.

try statement

- The source code catching an exception must surround the call with the try statement:

```
try {
  // this section contains commands that are
  // source of exception throwing
} catch (Exception e) {
  // caught exception is handled here,
  // it is accessible as variable with name e
}
```

try statement

1. Exception thrown from this method

```
try {
  exceptionalFilter.filterInputFile(fileName);
  //some following code
} catch(IOException e) {
    System.out.println("Unable to process file  "
     + fileName + "exception thrown: "
     + e.getMessage());
}
```

2. Control moves here

Catching multiple exceptions

```
try {
  // block of code that should throw exceptions
  FileInputStream fis = new FileInputStream("input.txt");
  // file processing
} catch (EOFException e) {
  // Take action on an end-of-file exception.
} catch (FileNotFoundException e) {
  // Take action on a file-not-found exception.
}
```

Handling different exceptions by same code block

```
try {
  // block of code that should throw exceptions
  FileInputStream fis = new FileInputStream("input.txt");
  // file processing
} catch (EOFException | FileNotFoundException e) {
  // Take action on an end-of-file and
  // file-not-found exception.
}
```

Finally clause

- The finally clause is executed even if a return statement is executed in the try or catch clauses.
- An uncaught or propagated exception still exits via the finally clause.

```
try {
  //Protect one or more statements here.
} catch (Exception e) {
  //Report and recover from the exception here.
} finally {
  //Perform any actions here common to whether or not
  //an exception is thrown.
}
```

Exception conversion

- Used when we need propagate a different exception type from a method
- Often used to the conversion into the RuntimeException or another unchecked exception
- Add useful information what's wrong
- Good practices is wrap original exception (use it as constructor parameter)

```
try {
  // ...
} catch (IOException e) {
  throw new MyException(e);
}
```

Features of exceptions

- methods:
  - getMessage
  - toString
  - printStackTrace
  - printStackTrace(PrintStream)

Defining new exception types

- Extend **RuntimeException** for an unchecked or **Exception** for a checked exception.
- We use our exception type to improve diagnostic information.
  - It can contain additional reporting and/or recovery information.

Defining new exception types in an action

```
public class NotConnectedException extends Exception {

  private String host;

  public NotConnectedException(String address) {
    super("Host: " + address + " is not accesible");
    host = address;
  }

  public String getHost() {
    return host;
  }

  @Override
  public String toString() {
    return "NotConnectedException [host=" + host
     + ", getMessage()=" + getMessage() + "]";
  }
}
```

Error recovery

- Clients should take description of error notifications.
  - After method calling, it checks the method return values.
  - Exceptions **should not** be ‘ignored’.
- Client code usually attempts to recover.
  - It is often implemented in a loop.

Attempting recovery

```
// Try to connect to server.
boolean successful = false;
int attempts = 0;
do {
    try {
        server.connect();
        successful = true;
    }
    catch(TimeoutException e) {
        System.out.println("Unable connect to " + server);
        attempts++;
        if(attempts < MAX_ATTEMPTS) {
            server = getAlternativeServer(server);
        }
    }
} while(!successful && attempts < MAX_ATTEMPTS);
if(!successful) {
    //Report the probslem and give up;
}
```

Enhanced syntax of “try” - motivation

```
BufferedReader reader = null;
try {
  reader = new BufferedReader(new FileReader("filename"));
  reader.lines();
} catch (FileNotFoundException e) {
  // the specified file could not be found
} catch (IOException e) {
  // something went wrong with reading
} finally {
  try {
    if (reader != null)
      reader.close();
  } catch (IOException e) {
    // something went wrong with closing
  }
}
```

Enhanced syntax of “try”

```
try (BufferedReader reader =
 new BufferedReader(new FileReader("filename"))) {
  String line = null;
  while (null != (line = reader.readLine())) {
    // do something with line
  }
} catch (FileNotFoundException e) {
  // the specified file could not be found
} catch (IOException e) {
  // something went wrong with reading
}
```

Enhanced syntax of “try” - usage

- It can be used on any class that implements interface AutoCloseable

```
public interface AutoCloseable {
    void close() throws Exception;
}
```

4th lecture – Java Collection Framework

- General Architecture
- Interfaces
- Implementations

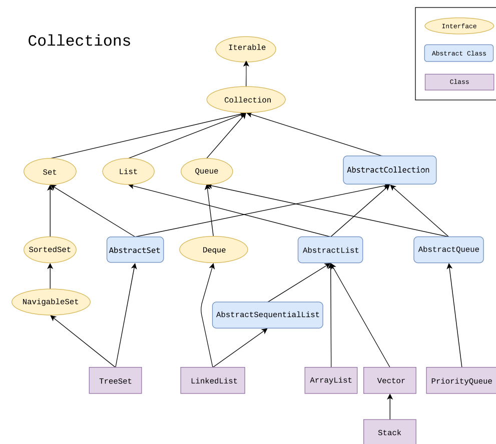
Collections in Java

- **Collection (container)** - objects that groups multiple elements into single unit.
- **Collections Framework**:
  - **Interfaces** – abstract data types representing collections
  - **Implementations** - concrete implementations of the interfaces – general, legacy, special-purpose, concurrent, wrapper, abstract
  - **Algorithms** - methods that perform useful computations (searching and sorting)

Collection types hierarchy

- Extended from:
  - java.util.Collection
  - java.util.Map
- java.util.Map – not true collection but offers collection-like manipulation

java.util.Collection - hierarchy



java.util.Collection

- base interface
- used to define a group of objects – allow manipulation, uniqueness and ordering is not defined for the interface
- methods:
  - add(E), addAll(Collection<E>), remove(E), removeAll(Collection<E>), clear(), retainAll(Collection<E>) – add/remove elements
  - size():int, isEmpty():boolean – check number of elements
  - contains (Object):boolean – check existence of element
  - iterator() - returns new iterator – enable browsing
  - toArray()

java.util.List

- ordered(defined index for every element) collection that may contain duplicate elements
- methods:
  - <extends Collection>
  - add(int,E), set(int,E), addAll(int,Collection<E>), get(int):E, remove(int):E – **add/remove elements to/from given position**
  - indexOf(Object):int, lastIndexOf(Object):int – **find position of given object**
  - listIterator():ListIterator – **return iterator that allows forward/backward browsing**

java.util.Set

- collection of elements that does not contain duplicates
- methods:
  - <extends Collection>
  - add(E):boolean, addAll(Collection<E>), contains(Object):boolean – added constraints to inherited methods

java.util.SortedSet

- set of elements where is defined ordering (index for items are not defined)
- methods:
  - <extends Set>
  - comparator(): Comparator<E>
  - subSet(E, E): SortedSet<E>
  - headSet(E): SortedSet<E>
  - tailSet(E): SortedSet<E>
  - first(): E
  - last(): E

java.util.Queue

- Queue is a list of elements with a first in first out ordering.
- When you enqueue an element, it adds it to the end of the list.
- When you dequeue an element, it returns the element at the front of the list and removes that element from the list.
- methods:
  - <extends Collection>
  - add(E), offer(E) – enqueue
  - remove(): E, poll(): E – dequeue
  - element():E, peek():E – retrieves but not remove

java.util.Deque

- double ended queue,
- enables enqueue to the start and dequeue from the end,
- provides stack functionality
- methods:
  - <extends Queue>
  - addLast/addFirst; getLast/getFirst – manipulation with end or beginning
  - push(E), pop(): E
  - descendingIterator(): Iterator<E>

Choose type of collection

- choose more general type.
- Iterable – only browsing (.. .and remove by iterator).
- others – modification (add, remove), provide size information, check existence of elements.

	Ordered	Indexed	Unique	FIFO	LIFO
Collection					
List	Y	Y			
Queue	Y			Y	
Deque	Y			Y	Y
Set			Y		
OrderdSet	Y		Y		
Map		Y – by key	Y – only key		

java.util.Map

- is a collection that links a key to a value.
- cannot contains duplicates of key – each key can only exists once and can only link to a single value.
- for key and value could be used any type

```
Map<KeyType, ValueType> myMap;
```

java.util.Map - example

- map String → Color

```
Map<String, Color> fruit2color = new HashMap<>();
```

- insert pairs

```
fruit2color.put("Apple", Color.RED);
fruit2color.put("Banana", Color.YELLOW);
fruit2color.put("Mellone", Color.GREEN);
```

- get value for a specific key

```
Color colorOfBanana = fruit2color.get("Banana");
Color colorOfApple = fruit2color.get("Apple");
```

java.util.Map – another methods

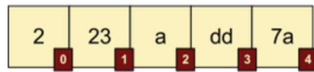
- containsKey(Object): boolean
- containsValue(Object): boolean
- keySet(): Set<K>
- values(): Collection<V>
- entrySet: Set<Entry<K,V>>
- remove(Object): V
- size(): int

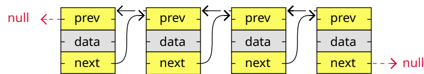
Collection Implementations

Interface	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	HashSet		TreeSet		LinkedHashSet
Collection	HashSet	ArrayList	TreeSet	LinkedList	LinkedHashSet
List		ArrayList		LinkedList	
Queue		ArrayDeque		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

List implementation: ArrayList vs LinkedList

- **Definition:**
  - **ArrayList**: A resizable array implementation of the List interface.
  - **LinkedList**: A doubly-linked list implementation of the List and Deque interfaces.
- **Performance:**
  - **ArrayList** has a faster average time for accessing elements as it uses an index-based system.
  - **LinkedList** has a faster average time for adding and removing elements.





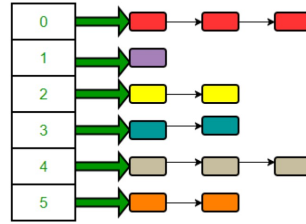
- **Memory Usage:**
  - **ArrayList** uses less memory as it holds only data.
  - **LinkedList** uses more memory as it holds data and two references for neighbor nodes.
- **Use Case:**
  - Use **ArrayList** when you have a fixed-size list, and you know the size won't change.
  - Use **LinkedList** when you have to change the list size frequently by adding or removing elements.
- **Syntax:**

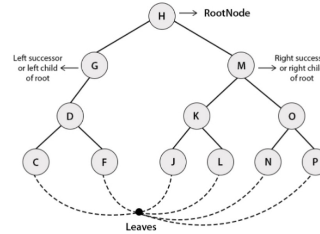
```
List<String> arrayList =
  new ArrayList<>();
List<String> linkedList =
  new LinkedList<>();
```

Map implementation: HashMap vs TreeMap

- **Definition:**
  - **HashMap**: Part of Java's collection since Java 1.2, provides the basic implementation of Map interface by hash table
  - **TreeMap**: A Red-Black tree based NavigableMap implementation, sorted according to the natural ordering of its keys.
- **Performance:**
  - **HashMap** generally offers constant time performance for the basic operations — get and put.
  - **TreeMap** guarantees log(n) time cost for the containsKey, get, put, and remove operations.
- **Ordering:**
  - **HashMap** does not maintain any order of its keys.
  - **TreeMap** maintains ascending order of its keys.
- **Null Keys and Values:**
  - **HashMap** allows one null key and multiple null values.
  - **TreeMap** does not allow null keys but may contain multiple null values.
- **Use Case:**
  - Use **HashMap** when you do not need sorted keys, and you need better performance.
  - Use **TreeMap** when you need sorted keys, and you can compromise on performance for ordering.
- **Syntax:**

```
Map<String, String> hashMap = new HashMap<>();
Map<String, String> treeMap = new TreeMap<>();
```





Set implementation: HashSet, TreeSet

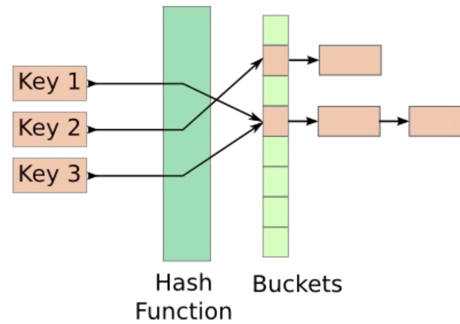
- It is implemented by HashMap / TreeMap

Requirements of using hash tables

- classes of objects (**value** objects) stored in hash tables:
  - objects stored in HashSet
  - keys used with HashMap
- should correctly override:
  - hashCode
  - equals (important also for comparison).
- **value** object – instances where their identity is not important but their state – String, Date, Money, Fraction, ComplexNumber…
-

hashCode method

- provide **Hash Function** for given object
- for two objects representing the same value **must** return same result
- for two objects representing different values **should** return different results – collisions are sometimes necessary
-



equals method

- used for **distinguishing between object** targeting the same bucket
- for two objects representing the same value must return true
- for two objects representing different values must return false
- used also in implementation of method contains (declared in Collection)
-