Variables

Public, Private and Protected

```
// accessible from outside the class.
public

// not accessible outside of the class
private

// not accessible outside of its own package, with exception from subclasses of
its class (childs)
protected
```

Static and Final

Theory

- **Static (Class) fields** belongs to the class, not to any object of the class;
- Static fields can be considered as fields that belong to all the objects of the class (black board)!
- **Instance fields** belongs to the object of the class;
- Instance fields are hot shared between the objects of the class!
- **Static (Class) methods** are those methods that do not have an implicit parameter; i.e., they do not operate on objects;
- call a static method by its Object e.g `Math.sqrt(4.5);` or `MyClass.MY_INT`
- Too many static methods are a sign of too little OOP!

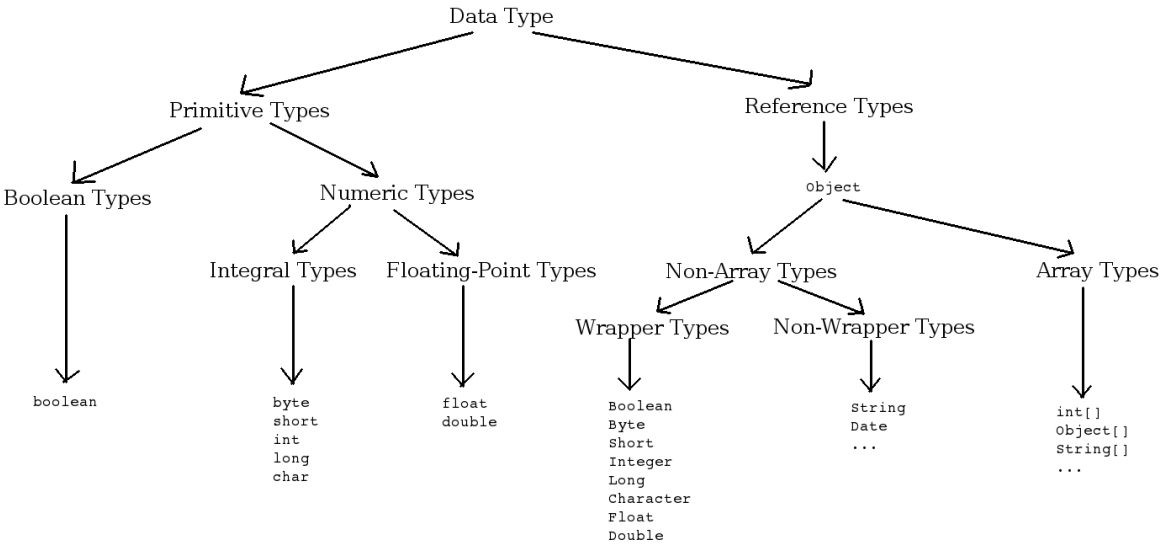
Rules

- Instance and static fields **can** be used from non-static methods;
- Static fields **can** be used from static methods;
- Instance fields **cannot** be used from static methods.

```
// a static value that does not change and isn't bound to object init.
public static final int MY_INT = 0;

// a static value that is not bound to object init.
public static int MY_INT = 0;

// Can only be assigned once, and can not be changed once assigned.
public final MY_INT = 0;

// The same static principle is used for methods.
public static void myFunction() {
    // TODO: add logic
}

// Final method means that the method can not be overridden by subclasses.
public final void myFunction() {
    // TODO: Add logic
}
```

Data Types

Data Type Tree



Type Casting

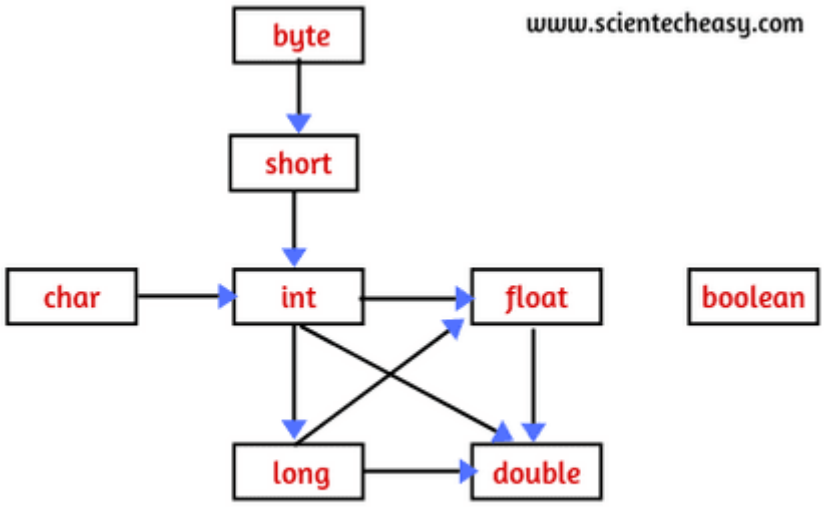


Fig: Automatic type conversion that Java allows.

1. If **byte, short, and int** are used in a mathematical expression, Java always **converts the result into an int**.
2. If a single **long** is used in the expression, **the whole expression is converted to long**.
3. If a **float** operand is used in an expression, **the whole expression is converted to float**.
4. If any operand is **double**, **the result is promoted to double**.
5. **Boolean** values **cannot be converted to another type**.
6. Conversion from **float to int causes truncation of the fractional part which represents the loss of precision. Java does not allow this**.
7. Conversion from **double to float causes rounding of digits that may cause some of the value's precision to be lost**.
8. Conversion from **long to int is also not possible**. It causes dropping of the excess higher order bits.

Widening Casting

Meaning: converting a smaller type to a larger type size. (automatically)
`byte -> short -> char -> int -> long -> float -> double`

```
int myInt = 9;
double myDouble = myInt; // Automatically casting int to double.

System.out.println(myInt); // 9
System.out.println(myDouble); // 9.0
```

Narrowing Casting

Meaning: converting a larger type to a smaller size type (manually)
`double -> float -> long -> int -> char -> short -> byte`

```
double myDouble = 9.78;
int myInt = (int) myDouble; // Manually casting double to int.

System.out.println(myDouble); // 9.78
System.out.println(myInt); // 9
```

Parsing from String

```
// Output: 1
int intNum = Integer.parseInt("1");
// Output: 2.0
double doubNum = Double.parseDouble("2.0");
// Output: 2.0
float flNum = Float.parseFloat("2");
String IntToStrNum = intNum.toString();
String DoubToStrNum = doubNum.toString();
String FlToStrNum = flNum.toString();
```

Can throw `Java.lang.NumberFormatException: empty String` or `java.lang.NullPointerException`.

Objects

```
Object name = new Object(parameters);
MyClass otherName = new MyClass(parameters);

// Creates and returns a copy of this object.
name.clone();

// Returns a boolean that indiciates whether some other object is "equal to"
this one.
name.equals(Object obj);

// Returns the runtime class of this Object.
name.getClass();

// Returns a string representation of the Object.
name.toString();
```

```
// Returns the name of the class as a String.
name.getName();

// Returns a hash value that is used to search object in a collection.
// Override of this method need to be done such that for every object we
generate a unique number.
// e.g Student class we can return studentNr from hashCode() method as it is
unique.
name.hashCode();
```

Comparable & Comparator

Comparable

utilizes `compareTo()` method from the `Comparable` interface.

```
class Movie implements Comparable<Movie> {
    private double rating;
    private String name;
    private int year;

    public int compareTo(Movie m) {
        return this.year - m.year;
    }
}

----------------------------------------
Main.java
----------------------------------------
ArrayList<Movie> molistvies = new ArrayList<Movie>();
list.add(new Movie("Force Awakens", 8.3, 2015));
list.add(new Movie("Star Wars", 8.7, 1977));
list.add(new Movie("Empire Strikes Back", 8.8, 1980));
list.add(new Movie("Return of the Jedi", 8.4, 1983));

// Uses the compareTo method to sort.
Collections.sort(list);

// Oldest on top, newest on bottom.
Star Wars 8.7 1977
Empire Strikes Back 8.8 1980
Return of the Jedi 8.4 1983
Force Awakens 8.3 2015
```

Comparator

- External to the element type we are comparing.
- Seperate class that implements Comparator to compare by different members.
- `Collections.sort()` also accepts Comparator as parameter.
- Opens `compare` method

3 things to do:

1. Create a class that implements Comparator
2. Make an instance of Comparator class

3. call overloaded `sort()`, providing both the list and Comparator.

```
class Movie implements Comparable<Movie> {
    private double rating;
    private String name;
    private int year;

    public int compareTo(Movie m) {
        return this.year - m.year;
    }
}

---------------------------------------
RatingComparer.java
---------------------------------------
class RatingComparer implements Comparator<Movie> {
    public int compare(Movie m1, Movie m2) {
        if (m1.getRating() < m2.getRating()) return -1;
        if (m1.getRating > m2.getRating()) return 1;

        return 0;
    }
}
---------------------------------------
NameComparer.java
---------------------------------------
class NameComparer implements Comparator<Movie> {
    public int compare(Movie m1, Movie m2) {
        return m1.getName.compareTo(m2.getName());
    }
}
---------------------------------------
Main.java
---------------------------------------
ArrayList<Movie> list = new ArrayList<Movie>();
list.add(new Movie("Force Awakens", 8.3, 2015));
list.add(new Movie("Star Wars", 8.7, 1977));
list.add(new Movie("Empire Strikes Back", 8.8, 1980));
list.add(new Movie("Return of the Jedi", 8.4, 1983));

RatingComparer rc = new RatingComparer();
NameComparare nc = new NameComparer();

// Sort using RatingComparer
Collections.sort(list, rc);

// Lowest on top, highest on bottom (Rating)
8.3 Force Awakens 2015
8.4 Return of the Jedi 1983
8.7 Star Wars 1977
8.8 Empire Strikes Back 1980

// Sort using NameComparer
Collections.sort(list, nc);

// Closest name to A on top, closest to Z on bottom.
Empire Strikes Back 8.8 1980
```

```
Force Awakens 8.3 2015
Return of the Jedi 8.4 1983
Star Wars 8.7 1977

// Obviously this would sort by year again
Collections.sort(list);
```

Interfaces

- Interfaces **specify** what a class **must do** and **not how**. It is the **blueprint** of the class.
- An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) `move()`. So it specifies a set of methods that the class has to implement.
- If a class implements an interface and **does not provide method bodies** for all functions specified in the interface, then the class **must be declared abstract**.
- `Comparator` and `Comparable` are two interface examples.
- We **can't create instance** (interface can't be instantiated) of interface but we can make reference of it that refers to the Object of its implementing class.
- An interface can extends another interface or interfaces
- All the methods are public and abstract. And all the fields are public, static, and final.
- It is used to achieve multiple inheritance.
- It is used to achieve loose coupling.

```
interface Vehicle {
    // public, static and final
    final String brand = "Audi";

    // public and abstract
    void changeGear(int a);
    void speedUp(int a);
    void applyBrakes(int a);
}
---------------------------------------
Bicycle.java
---------------------------------------
class Bicycle implements Vehicle {
    // fields here

    @Override
    public void changeGear(int newGear) {
        gear = newGear;
    }

    @Override
    public void speedUp(int inc) {
        speed += inc;
    }

    @Override
    public void applyBrakes(int dec) {
        speed = speed - dec;
    }
}

---------------------------------------
```

```
Bike.java
---------------------------------------
class Bike implements Vehicle {
    // fields here

    @Override
    public void changeGear(int newGear) {
        gear = newGear;
    }

    @Override
    public void speedUp(int inc) {
        speed += inc;
    }

    @Override
    public void applyBrakes(int dec) {
        speed = speed - dec;
    }
}
---------------------------------------
Main.java
---------------------------------------
Bicycle bicycle = new Bicycle();
bicycle.changeGear(2);
bicycle.speedUp(3);
bicycle.applyBrakes(1);
// Bicycle speed 2 and gear 2

Bike bike = new Bike();
bike.changeGear(1);
bike.speedUp(4);
bike.applyBrakes(3);
// Bike speed 1 and gear 1
```

New as of JDK 8+

Default Keyword

- Java now provides the `default` keyword for interface.
- Suppose we need to add a new function in an existing interface.
- Obviously the old code will not work as the classes have not implemented those new functions. So with the help of default implementation, we will give a default body for the newly added functions. Then the old codes will still work.

```
interface In1
{
    final int a = 10;
    default void display()
    {
        System.out.println("hello");
    }
}

class TestClass implements In1
{
```

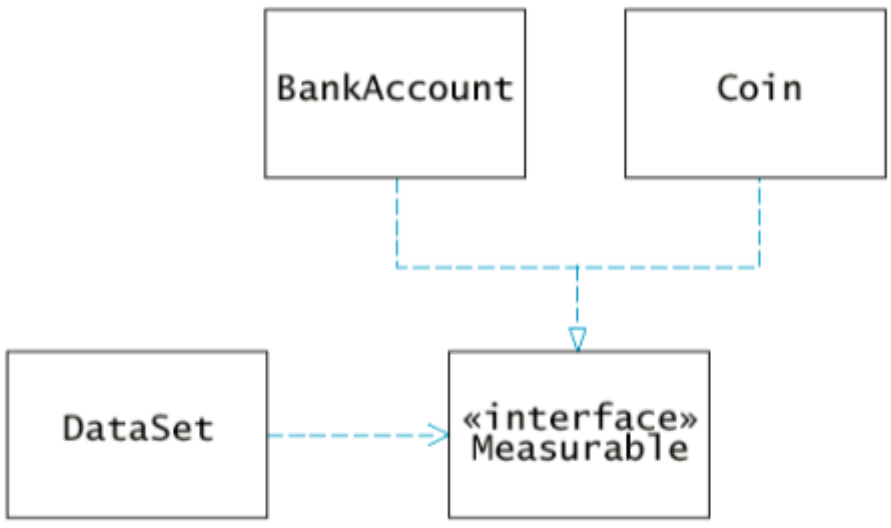
```
    // Driver Code
    public static void main (String[] args)
    {
        TestClass t = new TestClass();
        t.display();
    }
}

// Output: hello
```

Static Methods

- You can now define static methods in interface which can be called independently without an object.
- Note: these methods are not inherited.

```
interface In1
{
    final int a = 10;
    static void display()
    {
        System.out.println("hello");
    }
}

class TestClass implements In1
{
    // Driver Code
    public static void main (String[] args)
    {
        In1.display();
    }
}

// Output: hello
```

UML with Interface



Interface Casting

- You can type cast from **class to interface**.
- You can not cast from unrelated classes.
- If you cast from **Interface to Class** than you need a cast to convert from an interface type to a class type.

Example from Class to Interface:

```
// Interface: Vehicle
// Classes: Bike and Bicycle

// This is possible
Bike bike = new Bike();
Vehicle x = bike;

// This is not possible (unrelated type)
// Rectangle doesnt implement Vehicle
Measurable x = new Rectangle(5, 5, 5, 5);
```

Example from Interface to Class:

```
// Interface: Vehicle
// Classes: Bike

Bike bike = new Bike();
// speed: 1
bike.speedUp(1);

// a Method specific to the Bike class.
bike.getType();

Vehicle x = bike;
// speed: 3
x.speedUp(2);

// Error, as this method is not known within the interface.
x.getType();

// Can be fixed by casting:
Bike nb = (Bike)x;
// Now it'll work again
nb.getType();
```

Different Castings

- When casting number types, you lose information and you tell the compiler that you agree to the information loss: `int x = (int) 2.4;`
- When casting object types, you take a risk of causing an exception, and you tell the compiler that you agree to that risk: `Bike nb = (Bike)fastest;`

Controlling Casting

Use `instanceof`.

```
if (x instance of Bike) {
    Bike b = (Bike)x;
}
```

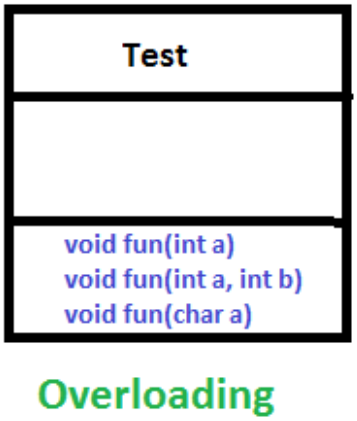
The `instanceof` operator returns `true` if the object is an instance of the class. Otherwise, it returns false.

Polymorphism

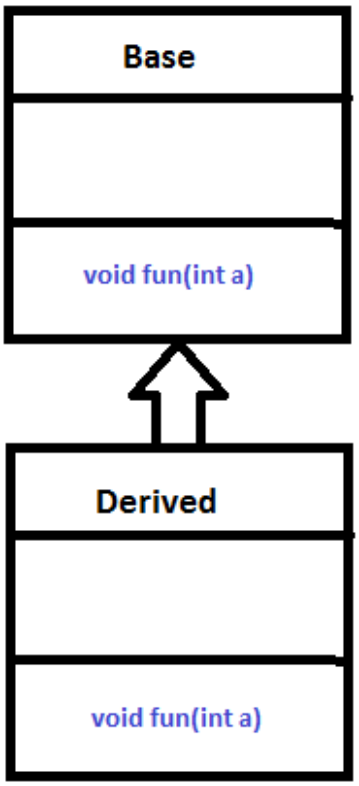
- The word polymorphism means **having many forms**.
- we can define polymorphism as the ability of a message to be **displayed** in **more than one form**.
- **Example**: A person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So the same person posses different behavior in different situations. This is called polymorphism.
- Polymorphism is considered one of the important features of Object-Oriented Programming. **Polymorphism allows us to perform a single action in different ways**.
- polymorphism allows you to define one interface and have multiple implementations.
- The word “poly” means many and “morphs” means forms, So it means many forms.

Compile Time Polymorphism

- It is also known as static polymorphism.
- This type of polymorphism is achieved by function overloading or operator overloading
- But **Java doesn’t support the Operator Overloading**.
- Early Binding



Overloading



Overriding

Method Overloading: When there are multiple functions with same name but different parameters then these functions are said to be **overloaded**. Functions can be overloaded by **change in number of arguments** or **change in type of arguments**.

```
class MultiplyFun {
    //  method with 2 parameters
    static int multiply(int a, int b) {
        // a = 2, b = 4 Output = 8
        return a*b;
    }

    // method with the same name but 3 parameters.
    static int multiply(int a, int b, int c) {
    // a = 2, b = 7, c = 3 Output = 42
    return a * b * c;
    }

    // method with the same name but 2 double parameters
    static double multiply(double a, double b) {
        // a = 5.5, b = 6.3 output = 34.65
        return a*b;
    }
}
```

Runtime Polymorphism

- It is also known as Dynamic Method Dispatch
- It is a process in which a function call to the overridden method is resolved at Runtime.
- This type of polymorphism is achieved by Method Overriding.
- Late Binding

Method overriding, on the other hand, occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

```
class Parent {
    void talk() {
        System.out.println("parent");
    }
}

class Daughter extends Parent {
    void talk() {
        System.out.println("daughter");
    }
}

class Son extends Parent {
    void talk() {
        System.out.println("son");
    }
}
---------------------------------------
Main.java
---------------------------------------
Parent a;

a = new Daughter();
a.talk(); // Output: daughter

a = new Son();
```

```
a.talk(); // output: son
```

Measurable Interface

```
public  class Dataset {
    private double sum;
    private Measurable maximum;
    private int count;

    public Measurable getMaximum() {
        return maximum;
    }

    public void add(Measurable x) {
        sum += x.getMeasure();

        if (count == 0 || maximum.getMeasure() < x.getMeasure()) {
            maximum = x;
        }

        count++;
    }
}

public class BankAccount implements Measurable {
    // methods and fields

    public double getMeasure() {
        return balance;
    }
}

public class Coin implements Measurable {
    // methods and fields

    public double getMeasure() {
        return value;
    }
}

public interface Measurable {
    double getMeasure();
}

----------------------------------------
Main.java
----------------------------------------
DataSet bankData = new DataSet();
bankData.add(new BankAccount(0));
11:       bankData.add(new BankAccount(10000));
12:       bankData.add(new BankAccount(2000));

// Output highest balance would be 10000.0

DataSet coinData = new DataSet();
coinData.add(new Coin(0.25, "quarter"));
23:       coinData.add(new Coin(0.1, "dime"));
```

```
24:         coinData.add(new Coin(0.05, "nickel"));

// Output highest coin value would be 0.25
```

UML

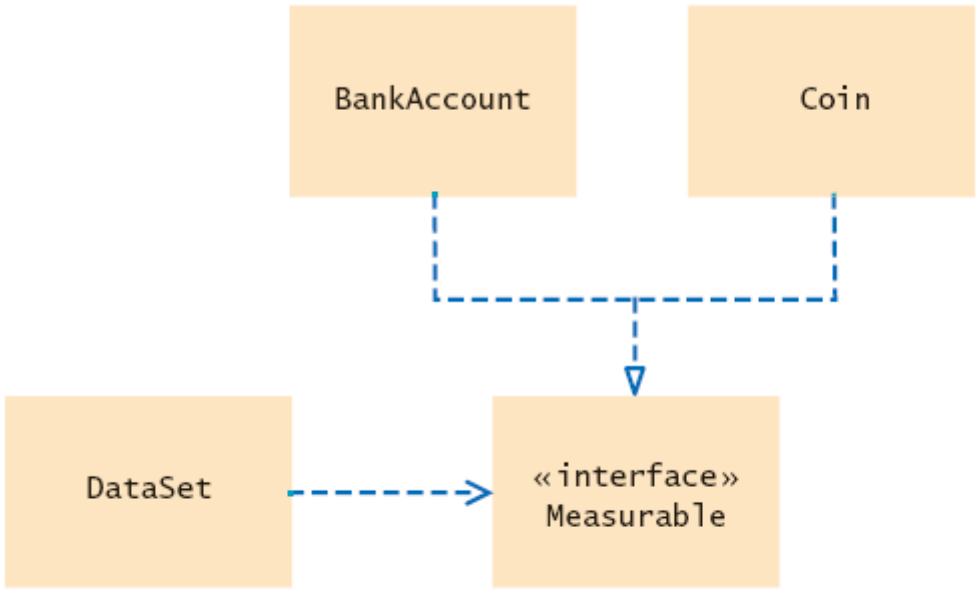


Figure 1 UML Diagram of the DataSet Class and the Classes That Implement the Measurable Interface

Converting between class and interface types

- You can convert from a class type to an interface type, provided the class implements the interface
- `BankAccount account = new BankAccount(10000);`
  `Measurable x = account; // OK`
- `Coin dime = new Coin(0.1, "dime");`
  `Measurable x = dime; // Also OK`
- Cannot convert between unrelated types
  `Measurable x = new Rectangle(5, 10, 20, 30); // ERROR`
  Because Rectangle doesn't implement Measurable

Casts

- Add coin objects to DataSet
  `DataSet coinData = new DataSet();`
  `coinData.add(new Coin(0.25, "quarter"));`
  `coinData.add(new Coin(0.1, "dime"));`
  `...`
  `Measurable max = coinData.getMaximum()`
- What can you do with it? It's not of type `Coin`
  `String name = max.getName(); // ERROR`
- You need a cast to convert from an interface type to a class type
- You know it's a coin, but the compiler doesn't. Apply a cast:
  `Coin maxCoin = (Coin) max;`
  `String name = maxCoin.getName();`
- If you are wrong and max isn't a coin, the compiler throws an exception

- Difference with casting numbers:
  When casting number types you agree to the information loss
  When casting object types you agree to that risk of causing an exception

Strategy Pattern (with Measurer example)

- Disadvantage of **Measurable** interface:
  - You can't force classes that aren't under your control to implement the interface (for example class Rectangle);
  - You can measure an object in only one way.
- Solution: let another object to carry out the measurements.

```
public interface Measurer {
    double measure(Object obj);
}

class RectangeAreaMeasurer implements Measurer {
    public double measure(Object obj) {
        Rectangle rect = (Rectangle)obj;
        return rect.getWidth() * rect.getHeight();
    }
}

public class Dataset {
    private double sum;
    private Object maximum;
    private int count;
    private Measurer measurer;

    public Dataset(Measurer measurer) {
        this.sum = 0;
        this.count = 0;
        this.maximum = null;
        this.measurer = measurer;
    }

    public void add(Object x) {
        sum += measurer.measure(x);
        if (count == 0 ||
        measurer.measure(maximum) < measurer.measure(x)) {
            maximum = x;
        }

        count++;
    }

    public double getAvg() {
        if (count == 0) return 0;

        return sum / count;
    }

    public Object getMaximum() {
        return maximum;
    }
}
```

```
---------------------------------------
Main.java
---------------------------------------
Measurer m = new RectangleMeasurer();
DataSet data = new DataSet(m);
data.add(new Rectangle(5, 10, 20, 30));
data.add(new Rectangle(10, 20, 30, 40));
data.add(new Rectangle(20, 30, 5, 10));

System.out.println("Average area = " + data.getAverage());
Rectangle max = (Rectangle) data.getMaximum();
System.out.println("Maximum area rectangle = " + max);
```

UML

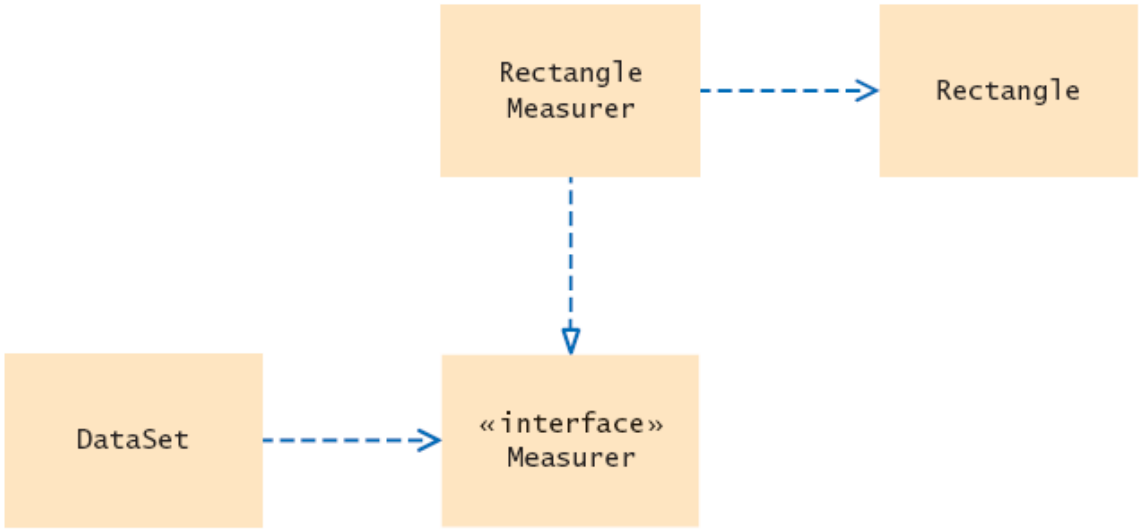


Figure 2 UML Diagram of the DataSet Class and the Measurer Interface

Nested Classes / Inner Classes

- A nested class is a class that is a member of another class.
- **Purpose:** To define an inner class whose scope is restricted to a single method or the methods of a single class
- Define a class within another class when the nested class makes sense only in the context of its enclosing class;
- A nested class has unlimited access to its enclosing class's members, even if they are declared `private`.

Inside a method

```
class OuterClassName {
  method signature {
     ...
     class InnerClassName {
        // methods
        // fields
     }
     ...
  }
  ...
}
```

Inside the class

```
class OuterClassName {
   // methods
   // fields
   accessSpecifier class InnerClassName {
      // methods
      // fields
   }
   ...
}
```

Static Nested Class

- A static nested class is specified with `static`:

```
class EnclosingClass {
    ...
    static  class StaticNestedClass {
        ...
    }
}
```

- A static nested class cannot refer directly to instance variables or methods defined in its enclosing class. It can use them only through an object reference.
- A static nested class can be independently instantiated without having creating an object of the outer class.

Non-Static (Inner) nested class

- A non-static nested class (inner class) is a nested class whose instance exists within an instance of its enclosing class and has access to the instance members of its enclosing instance.

```
class EnclosingClass {
    ...
    class NonStaticNestedClass {
        ...
    }
}
```

- A inner class cannot be independently instantiated without having creating an object of the outer class.

Anonymous Inner Class

- If you use a single object of an inner class, then use an anonymous inner class:

```
public static void main(String[] args) {
    Measurer m = new Measurer() {
        public double measure(Object obj) {
            Rectangle rect = (Rectangle)obj;
            return rect.getWidth() * rect.getHeight();
        }
    }

    Dataset data = new Dataset(m);
    data.add(new Rectangle(5, 10, 20, 30));     data.add(new Rectangle(10, 20,
30, 40));
    System.out.println("AverArea = " + data.getAverage());
}
```

Generics

- **Generics** are a facility of generic programming that were added to the Java programming language in 2004 within version J2SE 5.0.
- **Generics** were designed to extend Java's type system to allow *"a type or method to operate on objects of various types while providing compile-time type safety"*.

Motivation for generics

- The code below is compiled without error. However, it throws a **runtime exception** (`java.lang.ClassCastException`). This type of logic error can be detected during compile time.

```
ArrayList v = new ArrayList();
v.add("test");
Integer i = (Integer)v.get(0); // runtime error
```

- The above code fragment can be rewritten using generics as follows:

```
ArrayList<String> v = new ArrayList<String>();
v.add("test");
Integer i = v.get(0); // Compilation-time error.
```

- What we get is "**compile-time type safety**"; i.e. the errors appear during program development, ***not when they are used***.

Back to Generics again

- Java generic methods enable programmers to specify a set of related methods with a single method.
- Java generic classes enable programmers to specify a set of related types with a single class.
- This is achieved by using **type variables**.

- A **type variable** is an unqualified identifier; i.e. a variable whose value is a type.

Generic Classes

- A **class** is **generic** is generic if it declares one or more type variables. These type variables are known as the **type** parameters of the class.

```
public class A<T> {
    private T t;

    public T get() {
        return this.t;
    }

    public void set(T t) {
        this.t = t;
    }

    public static void main(String[] args) {
        A<String> a = new A<String>();
        a.set("CS2"); // valid
    }
}
```

Generic Methods

- A **method** is **generic** if it declares one or more type variables. These type variables are known as the formal **type** parameters of the method.

```
public static <T> boolean isEqual(A<T> g1, A<T> g2) {
    return g1.get().equals(g2.get());
}

public static void main(String[] args) {
    A<String> g1 = new A<String>();
    g1.set("CS2");
    A<String> g2 = new A<String>();
    g2.set("CS2");
    boolean equal = isEqual(g1, g2);
    // output result
}
```

Generic Interface

- An **interface** is **generic** if it declares one or more type variables. These type variables are known as its formal **type** parameters.

```
public interface MyInterface<T> {
    void myMethod(T t);
}

public class MyClass implements MyInterface<Integer> {
    public void myMethod(Integer t) {
        System.out.println(t);
    }
}
```

Processing Timer Events

- `javax.swing.Timer` objects generate timer events at fixed intervals;
- When a timer event occurs, the `Timer` object needs to notify some object called event listener;
- The class of the event-listener object has to implement the `ActionListener` interface:

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

- interface implementation:

```
class MyListener implements ActionListener {
    void actionPerformed(ActionEvent event) {
    // action logic here
    }
}
```

- Pass the reference of the listener to the `Timer` constructor and then start:

```
MyListener listener = new MyListener();
Timer t = new Timer(interval, listener);
t.start();
```

- Timer `t` calss the `actionPerformed` method of the listener object every `interval` milliseconds!

```
public class TimerTest {
    public static void main(String[] args) {
        class CountDown implements ActionListener {
            private int count;
            public CountDown(int count) {
                this.count = count;
            }

            public void actionPerformed(ActionEvent event) {
                if (count >= 0) {
                    // Output count
                }

                if (count == 0) {
                    // output liftoff
```

```
                }

                count--;
            }
        }

        CountDown listener = new CountDown(10);
        Timer t = new Timer(1000, listener);
        t.start();
    }
}
```

OOP

- To try to deal with the complexity of programs
- To apply principles of abstraction to simplify the tasks of writing, testing, maintaining and understanding complex programs
- To increase code reuse
    - to reuse classes developed for one application in other applications instead of writing new programs from scratch ("Why reinvent the wheel?")
- Inheritance is a major technique for realizing these objectives

Inheritance

- **Inheritance is a OOP principle. It allows us to extend existing classes by adding methods and fields.**
- The more general class is called a *super class*;
- The more specialized class that inherits from the superclass is called *subclass*;

```
public class BankAccount {
    private double balance;

    public BankAccount(double balance) {
        this.balance = balance;
    }

    public void deposit(double amount) {
        balance += amount;
    }

    public void withdraw(double amount) {
        balance = balance - amount;
    }

    public double getBalance() {
        return balance;
    }
}

public class SavingsAccount extends BankAccount {
    private double interestRate;

    public SavingsAccount(double rate) {
        interestRate = rate;
```

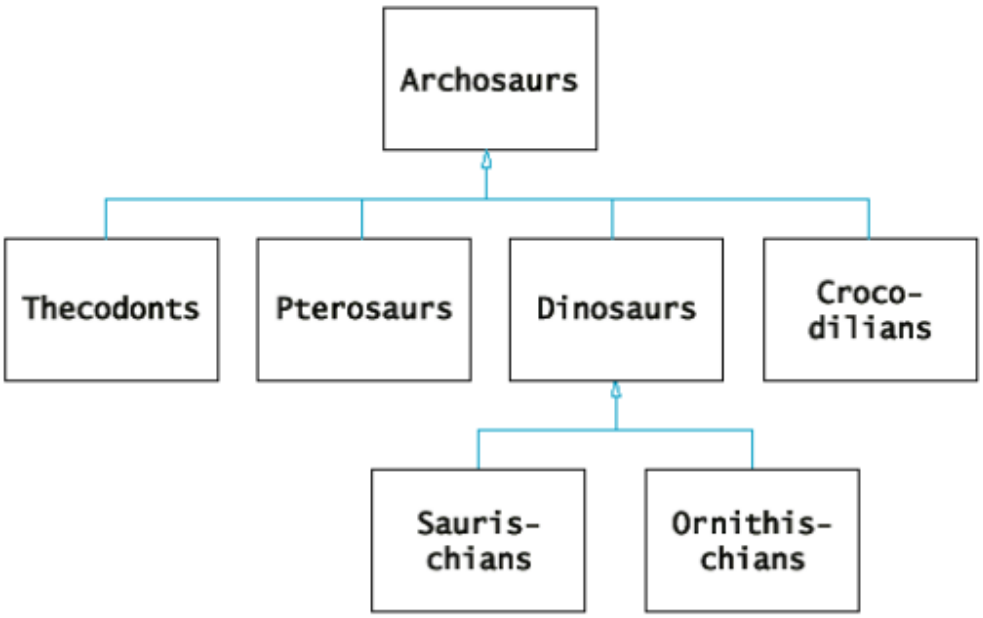
```
    }

    public void addInterest() {
        double interest = getBalance() * interestRate / 100;
        deposit(interest);
    }
}
```

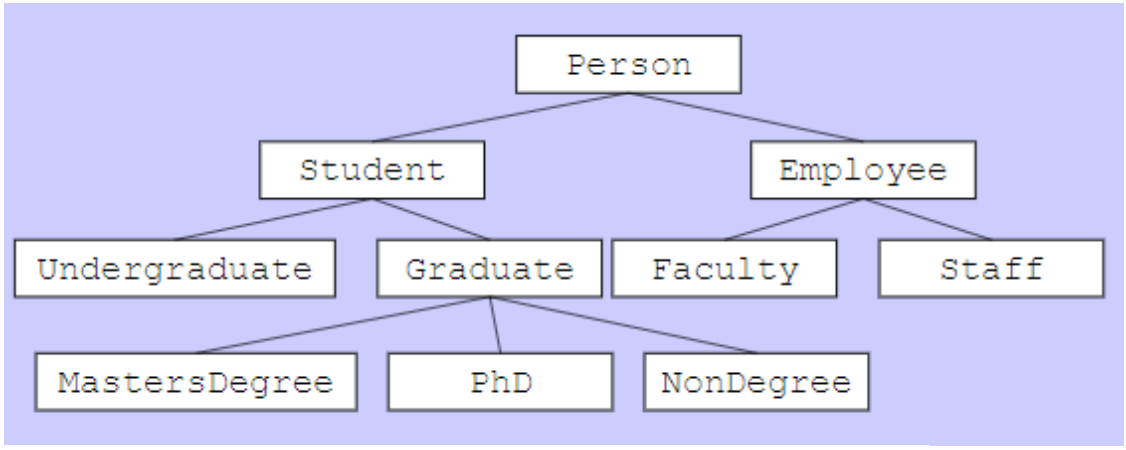
- Inheriting from a class differs from realizing an interface:
    - The **subclass** inherits behavior and the state of the **super class**;
    - An **interface** is not a class. It has not state and no **behavior**.
- Inheritance occurs between **classes** not **objects**;
- There are two types of inheritance:
    - **Single inheritance** is when children inherit characteristics from only oneparent. **Java employs single inheritance!**
    - **Multiple inheritance** is when children inherit characteristics from more than one parent.

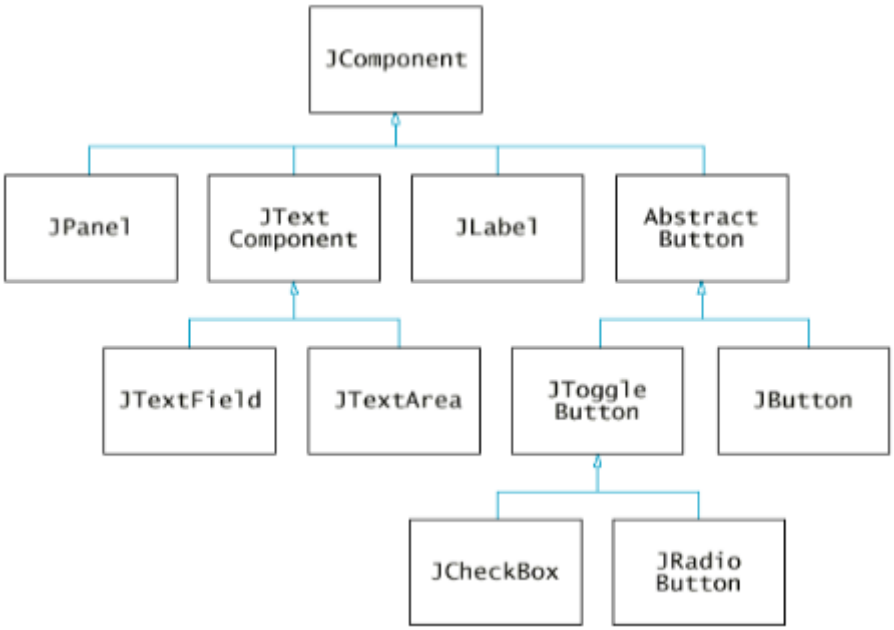
Derived Classes: Class Hierarchy

- The super classes can be used to implement specialized classes.
    - *For example: student, employee, faculty, and staff*

- Classes can be derived from the classes derived from the super class, etc., resulting in a **class hierarchy**.







Inheritance and methods

- When writing a subclass of superclass:
  - **You can override methods of the superclass**

    - **You can inherit methods of the superclass.**
    - **You can write new methods**
- **A method of a subclass overrides a method of a superclass if both methods have the same signature.**
- Assume that we have a class `CheckingAccount` with its own method `deposit`. `deposit` overrides the method `deposit` from `BankAccount`.

```
public class CheckingAccount extends BankAccont {
    private int transCount;
    public void deposit(double amount) {
        transCount++;
        super.deposit(amount);
    }
    ...
}
```

- **A subclass inherits a method from a superclass, if it does not have a method that overrides the superclass method**.
- The class `SavingsAccount` inherits the methods `getBalance` and `deposit` from `BankAccount`.

```
public class SavingsAccount extends BankAccont {
    public double interestRate;

    public SavingsAccount(double rate) {
        interestRate = rate;
    }

    public void addInterest() {
        double interest = getBalance() * interestRate / 100;
        deposit(interest);
    }
    ...
}
```

- A** subclass can have new methods of which the names or signatures differ those from the superclass. These methods can be applied only of the objects of the subclass.**
- The class `SavingsAccount` has a new method `addInterest`.

Inheritance and Fields

- When writing a subclass of superclass:
    - **You can inherit fields from the superclass**
    - **You can define new fields**
- You can inherit fields from the superclass. All fields from the superclass are automatically inherited.
- The field `balance` is inherited in `SavingsAccount`;
- But, since balance is `private` in `BankAccount`, `SavingsAccount` uses the method `getBalance` inherited from `BankAccount`.
- You can define new fields. In the class `SavingsAccount` the field `interestRate` is new.

- If a new field `balance` is defined in `CheckingAccount`,then an object of this class will have two fields with the same name. The newly defined field `shadows` the field in herited from the superclass.

Constructors in a subclass

- The subclass constructor can call a superclass constructor using the keyword **super**. The call has to be in the first line.
- If the subclass constructor doesn't call the super-class constructor, the default constructor of the **super-class** is called implicitly.

```
public class CheckingAccount extends BankAccount {
    private int transCount;
    public CheckingAccount(int initBalance) {
        super(initBalance);
        transCount = 0;
    }
}
```

- To call a constructor within a class use the keyword **this**.
- **this(0)** below calls constructor `CheckingAccount(int initialBalance)` with `equal` to `0`.

```
public class CheckingAccount extends BankAccount {
    private int transCount;
    public CheckingAccount(int initBalance) {
        super(initBalance);
        transCount = 0;
    }
    public CheckingAccount() {
        this(0);
    }
}
```

Call to an overriden method

- Use the keyword **super** to call a method from the superclass that was overridden in the subclass.
- `deposit` from `CheckingAccount` overrides `deposit` from `BankAccount`. `deposit` from `BankAccount` is called in `deposit` from `CheckingAccount` using super.
- the keyword `super` can be used for calling **superclass** non-overriden methods as well.

```
public class CheckingAccount extends BankAccount {
    private int transCount;
    public void deposit(double amount) {
        transCount++;
        super.deposit(amount);
    }
}
```

Convert from Subclass to Superclass

- Subclass references can be converted to superclass references:

```
SavingsAccount collegeFund = new SavingsAccount(10);
BankAccount anAccount = collegeFund;
Object anObject = collegeFund;
```

- Note that superclass references don't know the full story: `anAccount.addInterest(); // ERROR`

Abstract Classes

- An abstract class is a placeholder in a class hierarchy that represents a generic concept;
- An abstract class cannot be instantiated;
- We use the modifier **abstract** on the class header to declare a class as abstract;
- An abstract class often contains abstract methods (like an interface does), though it doesn't have to;

```
public abstract class BankAccount {
    public abstract void deductFees();
}
```

- The subclass of an abstract class must override the abstract methods of the parent, or it too will be considered abstract;
- An abstract method cannot be defined as **final** (because it must be overridden) or **static** (because it has no definition yet)
- The use of abstract classes is a design decision; it helps us establish common elements in a class that is to general to instantiate
- Usually **concrete** classes extend abstract ones, but the opposite is also possible.

References and abstract classes

- Suppose the following two classes were defined:

```
public abstract class Figure {}
public class Rectange extends Figure {}
```

- Are these instantiations correct?

```
Rectangle r = new Rectangle(...); // correct
Figure f = new Rectangle(...); // correct
Figure f = new Figure(...); // error
```

The cosmic Superclass (Object)

- **Object** is the superclass of all classes.
- Every class extends **Object**.
- Methods defined for **Object** are inherited by all classes.

ToString with Objects

- If no toString method is provided then it'll return the reference e.g `CheckingAccount@eee36c`

Equals

```
public boolean equals(Obj obj) {
    if (obj == null) return false;
    if (getClass() != obj.getClass()) return false;

    return true;
}
```

Clone

```
public Object clone() {
    BankAccount cloned = new BankAccount();
    cloned.balance = balance;
    return cloned;
}
```

WARNING:

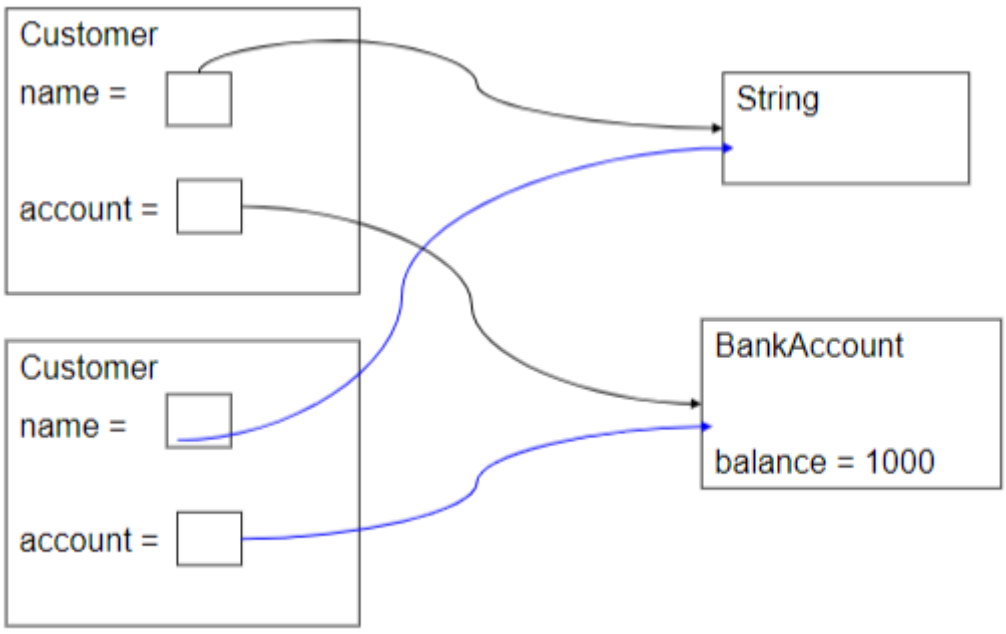
```
SavingsAccount  momsSavings = new SavingsAccount(0.5);
Object cloned = momsSavings.clone();
```

Constructs a new `BankAccount`, and not `SavingsAccount`.

Clone and Inheritance

- If the object contains references, the `Object.clone` method makes a **copy of the reference** not a clone of the object.

Shallow Copy



Graphical User Interface

Frameworks:

- Swing

- AWT (Abstract Windows Toolkit)
- JavaFX

Swing

Show an empty Frame

```
JFrame frame = new JFrame();
frame.setSize(300, 400);
frame.setTitle("A Title");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setVisible(true);
```

- Top left is point (0,0).

Drawing Shapes

```
public class RectangleComponent extends JComponent {
    public void paintComponent(Graphics g) {
        // recover graphics2d
        Graphics2D g2 = (Graphics2D)g;

        // draw a rectange
        Rectangle box = new Rectangle(5, 10, 20, 30);
        g2.draw(box);

        // Move rectangle 15 units to the right and 25 units down
        box.translate(15, 25);

        g2.draw(box);
    }
}

public class RectangleViewer {
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setSize(300, 400);
        frame.setTitle("Rectangles");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        RectangleComponent rc = new RectangleComponent();
        frame.add(rc);
        frame.setVisible(true);
    }
}
```

Graphical Classes: Double Version

- class `Point2D.Double` with constructor:
  - `Point2D.Double(double x, double y)`
- Class `Line2D.Double` with constructors:
  - `Line2D.Double(doubleX1, doubleY1, doubleX2, doubleY2)`
  - `Line2D.Double(Point2D p1, Point2D p2)`
- Class `Ellipse2D.Double` with constructor

- `Ellipse2D.Double(doublex, doubley, doublewidth, doubleheight)`

Class Color

- By default, all shapes are drawn in **black**
- To change the color, apply the method `setColor` on `g2` parameter with factual parameter an object of the class `Color`;
- The constructor of the class `Color` is:
  - `Color(float red, float green, float blue)` where **red** is the intensity of the **red** color, **green** is the intensity of the **green** color, and **blue** is the intensity of the **blue** color. **red**, **green**,and **blue** are **float** in the range `[0.0, 1.0]`.
- There are predefined colors: `Color.black`, `Color.green`, etc.
- Example: `g2.setColor(Color.red);`

BasicStroke

- To draw thicker lines, supply a different stroke object to `Graphics2D g2` parameter:
  - `g2.setStroke(new BasicStroke(4.0F));`

Drawing Strings

- To draw a string use method `drawString` of the class **Graphics2D**:
  - `g2.drawString(String s, float x, float y);` where **s** is the string to be drawn, and **x** and **y** are the **x** and **y** coordinates of the base point.
- Example: `g2.drawString("Applet", 50, 100);`

Class Font

- To change the font, apply the method `setFont` on **g2** parameter with factual parameter an object of the class **Font**;
- `Font(Stringname, intstyle, intsize)`
  - where **name** is the font name such as `Serif`, `SansSerif`, `Monspaced` etc.,style is an integer(`Font.PLAIN`, `Font.BOLD`, `Font.ITALIC`), and **size** is the point size.
- Example:

```
Font myFont= new Font("Serif", Font.BOLD, 36);
g2.setFont(myFont);
g2.drawString("Application", 50, 100);
```

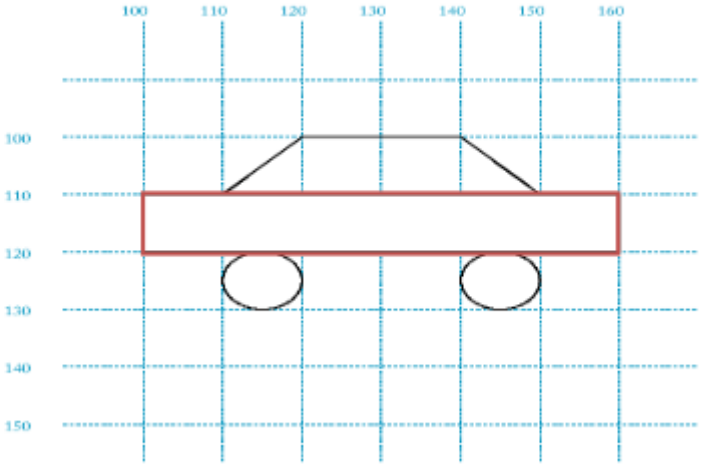
Frame with Two Cars

```
import javax.swing.JFrame;
```

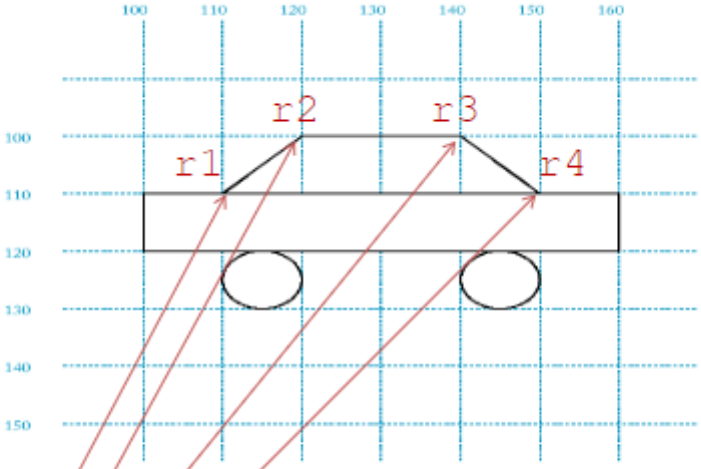
```
public class CarViewer {
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setSize(300, 400);
        frame.setTitle("Cars");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        CarComponent cc = new CarComponent();
        frame.add(cc);
        frame.setVisible(true);
    }
}

-------------

import javax.swing.JComponent;
import java.awt.Graphics;
import java.awt.Graphics2D;

public class CarComponent extends JComponent {
    public void paintComponent(Graphics g) {
        Graphics2D g2 = (Graphics2D)g;
        Car car1 = new Car(100, 100);
        Car car2 = new Car(150, 150);
        car1.draw(g2);
        car2.draw(g2);
    }
}

-------------

public class Car {
    private double xLeft;
    private double xTop;

    public Car(double x, double y) {
        xLeft = x;
        xTop = y;
    }

    public void  draw(Graphics2D g2) {
        Rectangle2D.Double body = new Rectangle2D.Double(xLeft, yTop + 10, 60,
10);
        Ellipse2D.DoublefrontTire=newEllipse2D.Double(xLeft+ 10,yTo p+
20,10,10);
        Ellipse2D.DoublerearTire=newEllipse2D.Double(xLeft+ 40,yTo p+ 20,10,10);
        Point2D.Doubler1 =newPoint2D.Double(xLeft+ 10,yTo p+ 10);
        Point2D.Doubler2 =newPoint2D.Double(xLeft+20,yTo p);
        Point2D.Doubler3 =newPoint2D.Double(xLeft+ 40,yTo p);
        Point2D.Doubler4 =newPoint2D.Double(xLeft+ 50,yTo p+ 10);
        Line2D.DoublefrontWindshield=newLine2D.Double(r1, r2);
        Line2D.DoubleroofTop=newLine2D.Double(r2, r3);
        Line2D.DoublerearWindshield=newLine2D.Double(r3, r4);
        g2.draw(body);
        g2.draw(frontTire);
        g2.draw(rearTire);
        g2.draw(frontWindshield);
        g2.draw(roofTop);
```

```
        g2.draw(rearWindshield);
    }
}
```

```
xLeft = 100;
yTop = 100;
```

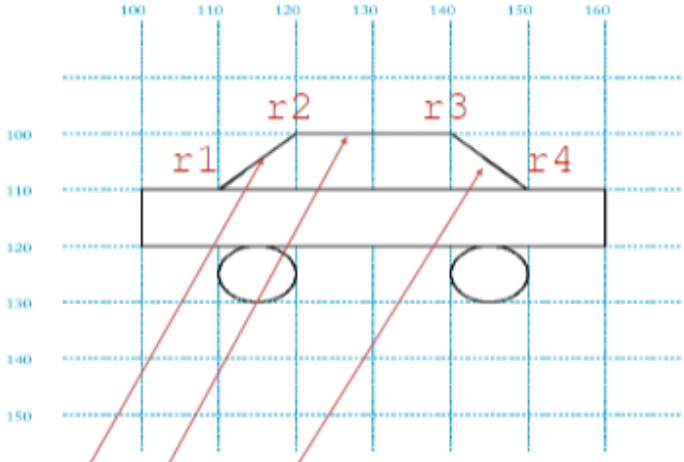



```
xLeft = 100;
yTop = 100;
```




```
public void draw(Graphics2D g2) { ...
    Point2D.Double r1 = new Point2D.Double(xLeft + 10, yTop + 10);
    Point2D.Double r2 = new Point2D.Double(xLeft+20, yTop);
    Point2D.Double r3 = new Point2D.Double(xLeft + 40, yTop);
    Point2D.Double r4 = new Point2D.Double(xLeft+ 50, yTop + 10); ... }
```

```
xLeft = 100,
yTop = 100;
```

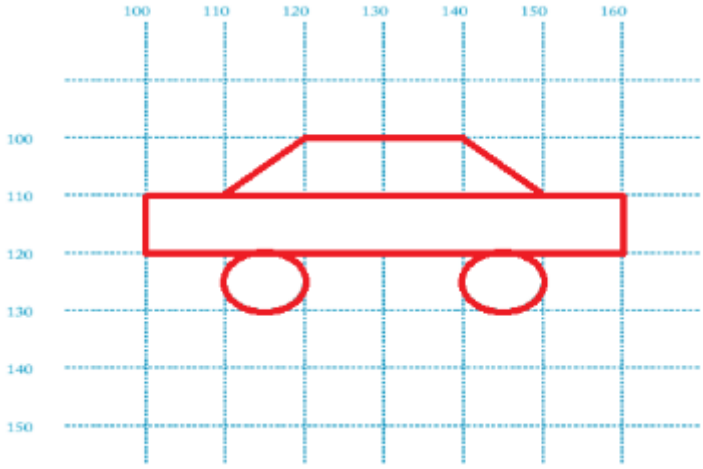



```
public void draw(Graphics2D g2) { ...
    Line2D.Double frontWindshield = new Line2D.Double(r1, r2);
    Line2D.Double roofTop = new Line2D.Double(r2, r3);
    Line2D.Double
```

```
... }
```

"Tuning" The car

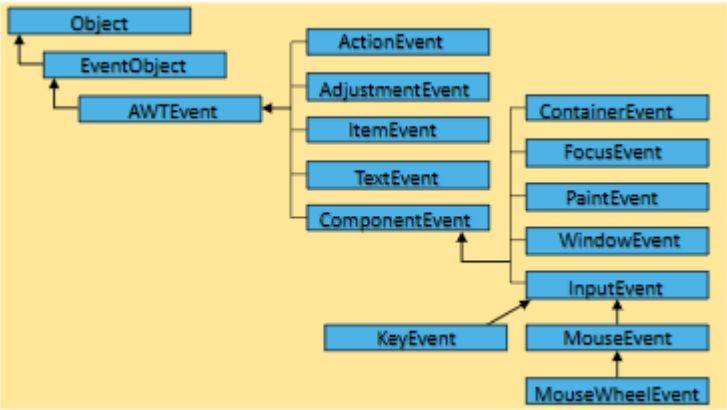
```
    public void draw(Graphics2D g2) {
        g2.setStroke(new BasicStroke(2));
        g2.setColor(Color.RED);
        g2.draw(body);
        g2.draw(frontTire);
    }
```

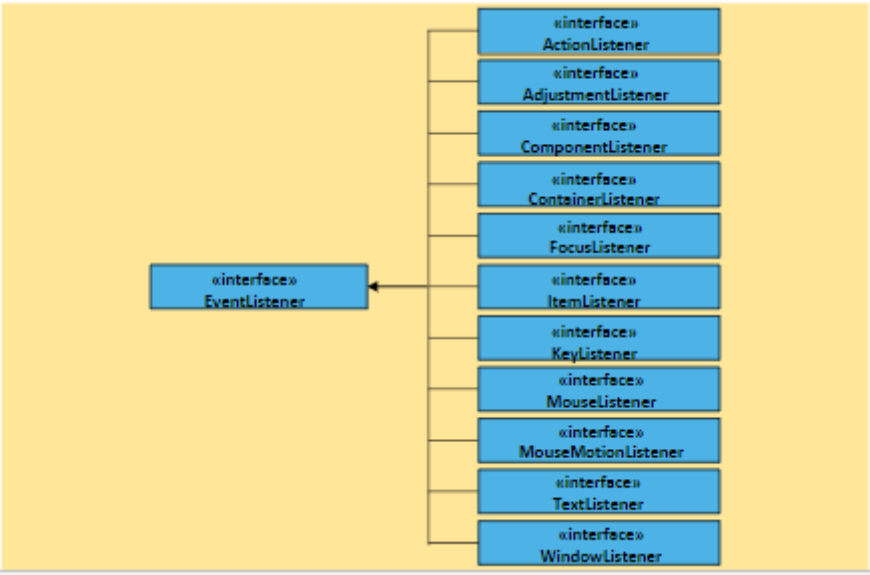


Reading Input

```
import javax.swing.JOptionPane;
class ReadingInput {
    public static void main(String[] args) {
        String intput = JOptionPane.showInputDialog("Enter Number: ");
        int count = Integer.parseInt(input);
        ...
        System.out.println("Number: " + input);
    }
}
```

Event Handling





Mouse Spy

```
import java.awt.events.*;

class MouseSpy implements MouseListener {
    public  void mousePressed(MouseEvent e) {
        sout(e.getX() + " " + e.getY());
    }

    public  void mouseRelease(MouseEvent e) {
        sout(e.getX() + " " + e.getY());
    }

    public void mouseClicked(...) {}
    public void mouseEntered(...) {}
    public void mouseExited(...) {}
}

-------
import javax.swing.*;

public c lass MouseSpyViewer extends JFrame {
    public MouseSpyViewer() {
        MouseSpy listener = new MouseSpy();
        addMouseListener(listener);
    }

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

```
        MouseSpyViewer frame = new MouseSpyViewer();
        frame.setSize(...);
        ...
        frame.setVisible(true);
    }
}
```

Mouse frame

```
public class MouseComponent extends JComponent {
    private Rectangle box;

    public MouseComponent() {
        box = new Rectangle(100, 100, 20, 30);

        class MousePressListener implements MouseListener {
            public void mousePressed(MouseEvent e) {
                int x = e.getX();
                int y = e.getY();
                box.setLocation(x, y);
                repaint();
            }

            public void mouseReleased(...){}
            public void mouseClicked(...){}
            public void mouseEntered(...){}
            public void mouseExited(...){}
        }

        MousePressListener listener = new MousePressListener();
        addMouseListener(listener);
    }
    public void paintComponent(Graphics g) {
        Graphics2D g2 = (Graphics2D)g;
        g2.draw(box);
    }
}

----
main {
    MouseComponent mc = new MouseComponent();
    JFrame frame = new JFrame();
    frame.add(mc);
}
```

Inner Class Listener

```
class MyClass {
    public MyClass() {
        class MyListener implements ListenerInterface {
            public void eventOccured(EventClass e) {
                // event actions go here!
            }
        }

        MyListener listener = new MyListener();
        anEventSource.addListener(listener);
    }
}
```

```
public class ButtonFrame extends JFRame {
    private JButton button;
    private JLabel label;

    class ClickListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            label.setText("clicked");
        }
    }
}
```

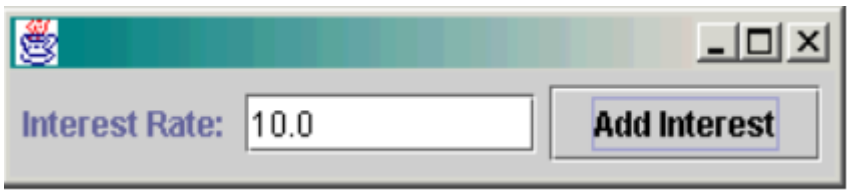
Adapter Class

- Adapter classesprovide the default implementation of listenerinterfaces
- If you inherit the adapter class, you will not be forced to provide the implementation of all the methods of listener interfaces
- So it saves time/code
- To avoid writing empty methods in your listeners, use the standard **MouseAdapter** class;
- This class implements the **MouseListener** interface, but all the methods are empty;
- Thus, when you extend your listener with the **MouseAdapter** class, you write only nonempty methods

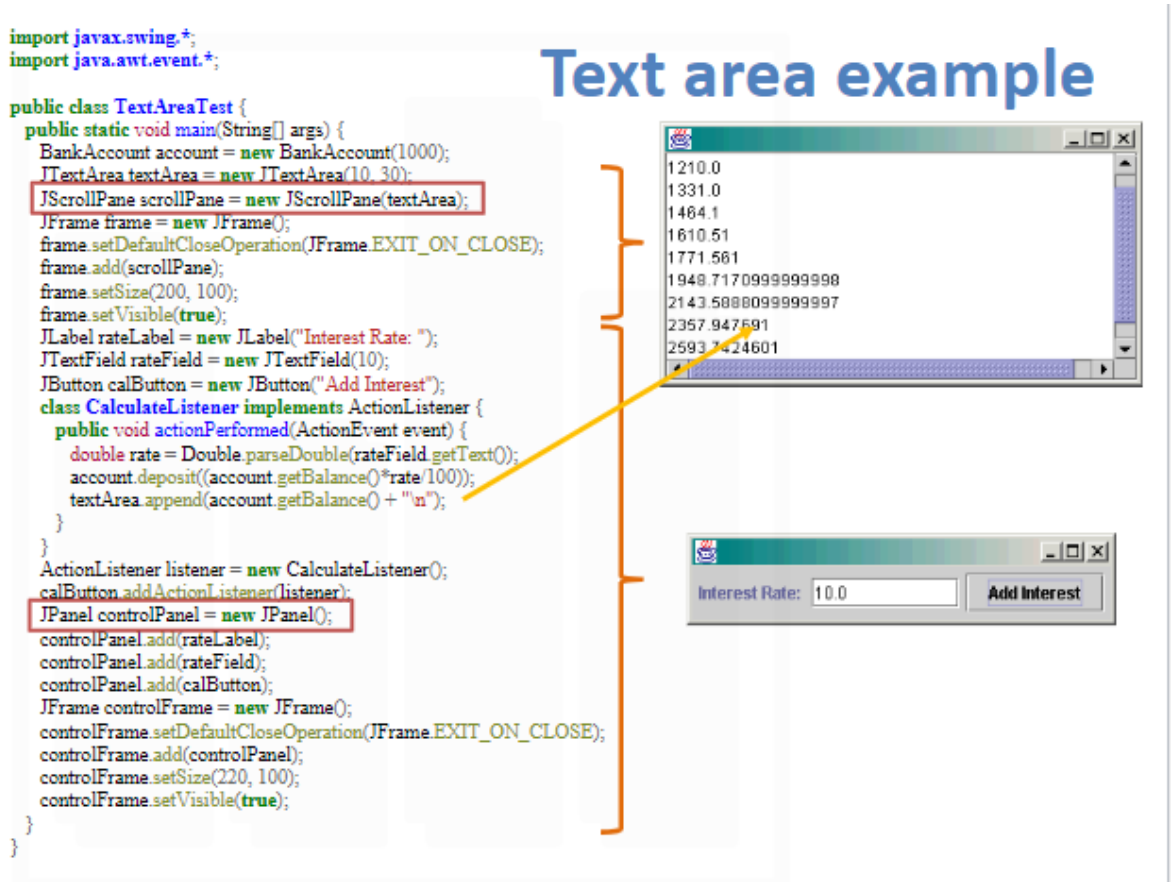
```
class MousePressListener extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
        int x = e.getX();
        int y = e.getY();
        box.setLocation(x, y);
        repaint();
    }
}
```

Component Examples

```
JTextFieldrateField= new JTextField(10);
JLabelxLabel= new JLabel("Interest Rate: ");
JButtoncalButton= new JButton("AddInterest");
```



Text Component



Layout Management

- Three useful layout managers: `flow layout`, `border layout`, `grid layout`.
- **JFrame**: framed window
  - Generally used as a window for hosting stand-alone applications, like an alert window or notification window
  - Contains the title bar
- **JPanel**: works as a container to host components
  - Can be considered as general container, which is used in case of complex or bigger functions which require grouping of different components together

Flow Layout

- Flow layout used by default by **JPanel**
- arranges components from left to right
- Starts a new row when no more room is left in current row.

Border Layout

- The border layout groups components into 5 areas (CENTER, NORTH, SOUTH, WEST, EAST)
- Border layout used by default by **JFrame**.
- Use in panel using `panel.add(component, BorderLayout.NORTH)`

Grid Layout

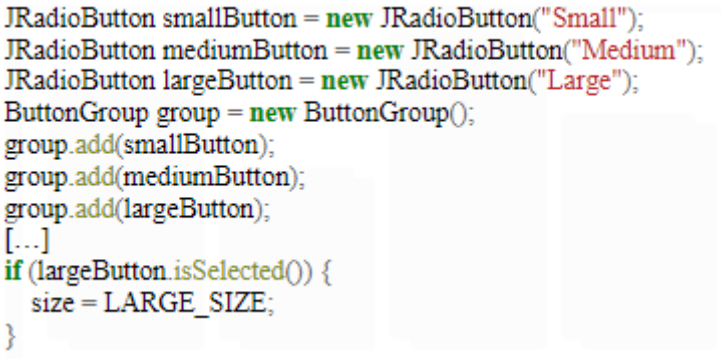
- The grid layout arranges components in a grid with a fixed number of rows and columns
- All components are resized so that they all have the same width and height
- Like the border layout, it expands each component to fill the entire allotted area
- If not desirable, you need to place each component inside a panel
- To create a grid layout, you supply the number of rows and columns in the constructor, then add the components row by row, left to right

```
JPanelbuttonPanel= new JPanel();
buttonPanel.setLayout(new GridLayout(4, 3));
buttonPanel.add(button1);
buttonPanel.add(button2);
buttonPanel.add(button3);
buttonPanel.add(button4);
```

Choices

- check box (selectbox)

Radio Buttons



```
JRadioButton smallButton = new JRadioButton("Small");
JRadioButton mediumButton = new JRadioButton("Medium");
JRadioButton largeButton = new JRadioButton("Large");
ButtonGroup group = new ButtonGroup();
group.add(smallButton);
group.add(mediumButton);
group.add(largeButton);
[...]
if (largeButton.isSelected()) {
   size = LARGE_SIZE;
}
```

Checkboxes

- A check box is a UI component with 2 states: **checked** and **unchecked**
- They are not exclusive
- Because check box settings do not exclude each other, you do not place a set of check boxes inside a button group
- You can use isSelectedmethod to find out whether a check box is currently checked or not

```
JCheckBoxitalicBox= new JCheckBox("Italic");
JCheckBoxboldBox= new JCheckBox("Bold");
```

Comboboxes

- If you have a lot of choices and little space you can use a combo box
- A combination of a list and a text field
- If the combo box is editable you can also type in your own selection
- To make a combo box editable, call the `setEditable` method

```
setEditablemethodJComboBoxcombo = new JComboBox();
combo.addItem("Serif");
combo.addItem(" SansSerif");
...
String select = (String) combo.getSelectedItem();
```

Button Groups

- `javax.swing.border.TitledBorder`: A class which implements an arbitrary border with the addition of a String title in a specified position and justification
- `javax.swing.border.EtchedBorder`: A class which implements a simple etched border which can either be etched-in or etched-out. If no highlight/shadow colors are initialized when the border is created, then these colors will be dynamically derived from the background color of the component argument passed into the `paintBorder()` method.

Menus

- At the top of the frame is a **menu bar** that contains the top-level menus
- Each menu is a collection of **menu items** and **submenus**

```
public class MyFrame extends JFrame {
   public MyFrame() {
      JMenuBar menuBar = new JMenuBar();
      setJMenuBar(menuBar);
      JMenu fileMenu = new JMenu("File");
      JMenu fontMenu = new JMenu("Font");
      menuBar.add(fileMenu);
      menuBar.add(fontMenu);
      JMenuItem exitItem = new JMenuItem("Exit");
      fileMenu.add(exitItem);
      ...
      JMenu styleMenu = new JMenu("Style");
      fontMenu.add(styleMenu);
      ...
   }
}
```

- When the user selects a menu item, the menu item sends an action event. Therefore, you must add a listener to each menu item

```
ActionListenerlistener = new ExitItemListener();
exitItem.addActionListener(listener);
```

- You add action listeners only to menu items, not to menus or the menu bar

- When the user clicks on a menu name and a submenu opens, no action event is sent

```
public JMenuItem createItem(final String name) {
// Global variables can be accessed from an inner class method
    private JLabel label;
    class MyItemListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            newName = name;
            setNewText();
        }
    }
    JMenuItem item = new JMenuItem(name);
    ActionListener listener = new MyItemListener();
    item.addActionListener(listener);
    return item;
}
public void setNewText() {
    label.setText(newName);
}
```

Timer Events

- The **Timer** class in the `javax.swing` package generates a sequence of action events, spaced at even time intervals
- This is useful whenever you want to send continuous updates to a component

1. When you use a timer you specify the frequency of the events and an object of a class that implements the **ActionListener** interface
2. Place whatever action you want to occur inside the **actionPerformed** method
3. Start the timer

```
class MyListener implements ActionListener {
  public void actionPerformed(ActionEvent event) {
     // Action that is executed at each timer event
  }
}
MyListener listener = new MyListener();
Timer t = new Timer(TIME, listener);  // TIME in milliseconds
t.start();
```

Mouse Events

- Mouse listeners are more complex than action listeners
- A mouse listener must implement the MouseListenerinterface, which contains the following 5 methods

```
public interface MouseListener{
    void mousePressed(MouseEventevent);
    void mouseReleased(MouseEventevent);
    void mouseClicked(MouseEventevent);
    void mouseEntered(MouseEventevent);
    void mouseExited(MouseEventevent);
}
```

- It often happens that a particular listener specifies actions only for one or two of the listener methods. Nevertheless, all 5 methods of the interface must be implemented.
    - You can also use **MouseAdapter**

Scanner

```java
import java.util.Scanner;

Scanner in = new Scanner(System.in);
int num = in.nextInt(); //Integer
String str = in.nextLine(); // String
boolean bo = in.nextBoolean(); // boolean
double doub = in.nextDouble(); // Double

String.out.println("String: " + str);
String.out.println("Integer: " + num);
String.out.println("Double: " + doub);
```

If the given input is wrong you will receive a `InputMismatchException` exception.

Exception Handling

Code

```
readFile()
 {
       openFile();
       determineFileSize();
       allocateMemory();
       readFileIntoMemory();
       closeFile();
  }
```

Possible Errors

- the file can't be opened
- the length of the file can't be determined
- enough memory can't be allocated
- the read fails
- the file can't be closed

Traditional approach

- Each method returns error code:

```
int readFile() {
    errorCode = 0;
    if (!openFile())
        errorCode = 1;
    else if (!determineFileSize())
        errorCode = 2;
    else if (!allocateMemory())
        errorCode = 3;
    else if (!readFileIntoMemory())
        errorCode = 4;
    else if (!closeFile())
        errorCode = 5;
    return errorCode;
}
```

Problems of Trad. Appr.

- Calling method may forget to check for error code
- Calling method may not know how to fix error
- Spaghetti code
- Programming for success is replaceed by programming for failure

Exceptions

- **Definition:** An exceptionis an event that occurs during the execution of a program that disruptsthe normal flow of instructions

```
void readFile() {
    try {
        openFile();
        determineFileSize();
        allocateMemory();
        readFileIntoMemory();
        closeFile();
    }
    catch (FileOpenFailed e) {}
    catch (FileSizeFailed e) {}
    catch (memoryAllFailed e) {}
    catch (ReadFailed e) {}
    catch (FileCloseFailed e) {}
    finally{//always executed}
}
```

Advantages of exceptions

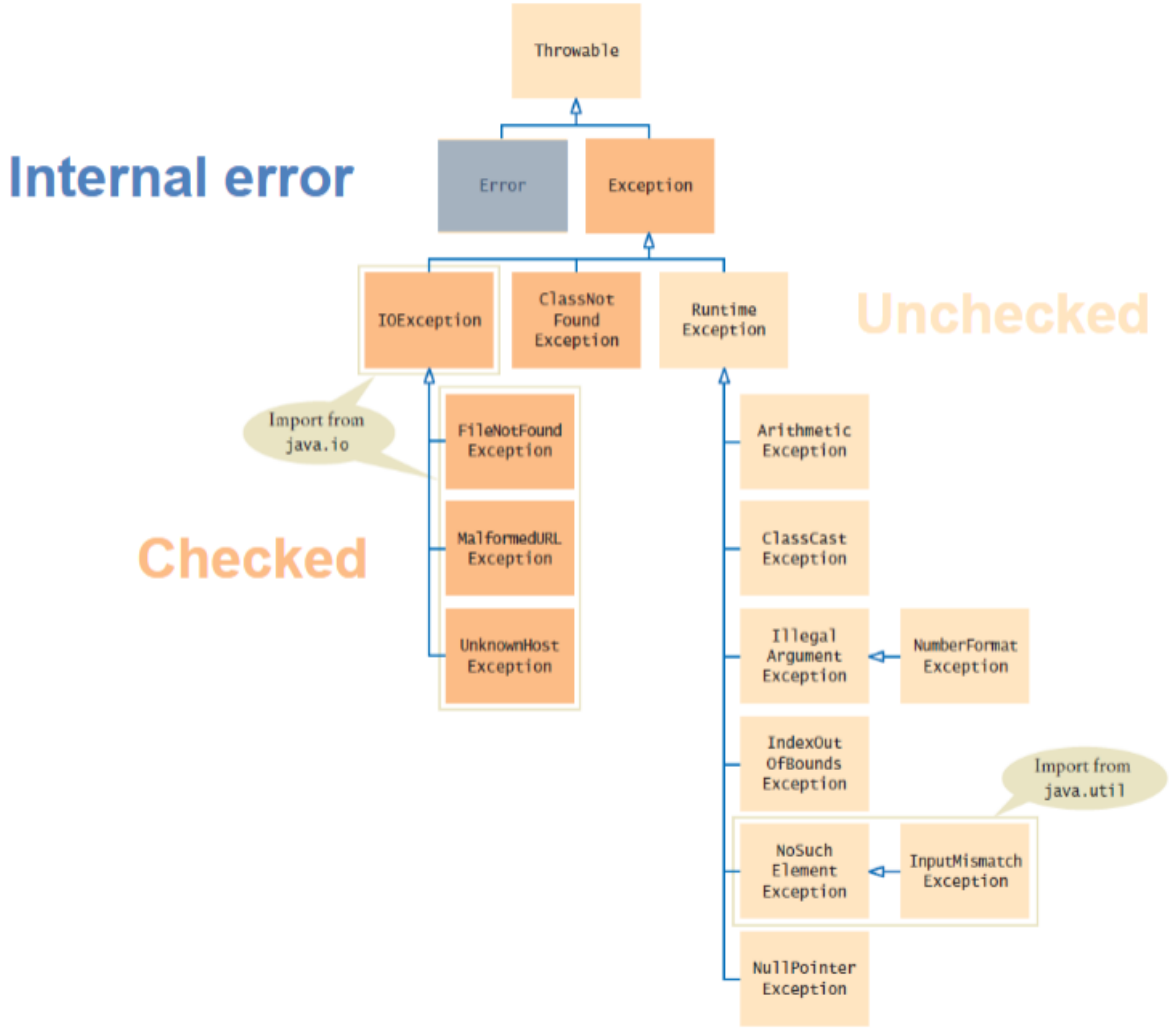
1. Separating error handling code from “regular" code
2. Propagating errors up the call stack
3. Grouping exceptions

The Java language requires that methods either catch or specify all exceptions that can be thrown within the scope of that method!!!

```
void readFileIntoMemory() throws readFailed, IOException
void openFile() throws fileOpenFailed, EOFException
```

Checked and Unchecked exceptions

- 3 catagories:
    - Internal error: fatal errors occuring rarely (`OutOfMemoryError`)
    - Checked (at compilation time): something was wrong for some external reason beyond our control (`FileNotFound`)
    - Unchecked Exception: Runtime Exception (`IndexOutOfBoundsException`)



Common build-in exceptions

- **ArithmeticException**: division by zero
- **ArrayIndexOutOfBoundsException**: array index does not exist
- **IOException**: failure during reading, writing and searching file (`FileNotFoundException`)
- **NoSuchFieldException**: class does not have the specified field (variable)
- **NoSuchMethodException**: particular method cannot be found
- **NullPointerException**: use an object reference that has the null value
- **NumberFormatException**: String parsed to any numerical value

- **StringIndexOutOfBoundsException**: string index does not exist

```
Scanner in = null;

try {
    in = new Scanner(System.in);
    System.out.println("How old are you?");
    String inputLine = in.next();
    int age = Integer.parseInt(inputLine);
    age++;
    System.out.println(age);
}
catch (NumberFormatException exception) {
    System.out.println("Input was not a number");
}
finally {
    if (in != null)
        in.close();
}
```

Designing exception types

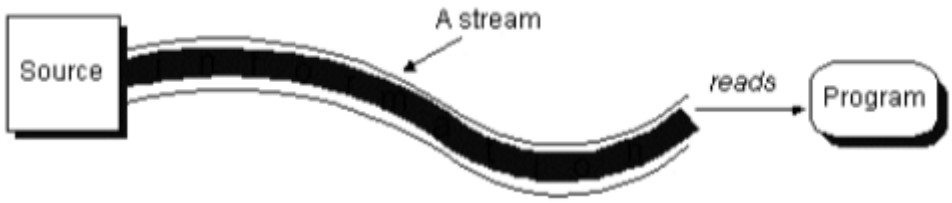
```
public class InsufficientFundsException
    extends RuntimeException
{
    public InsufficientFundsException()
    {

    }

    public InsufficientFundsException(String reason)
    {
        super(reason);
    }
}
```

```
public class BankAccount {
  public void withdraw(double amount)
    throws InsufficientFundsException
  {
      if (amount > balance) {
        throw new InsufficientFundsException("Amount exceeds balance");
      }
      balance = balance – amount;
  }
  ...
}
```

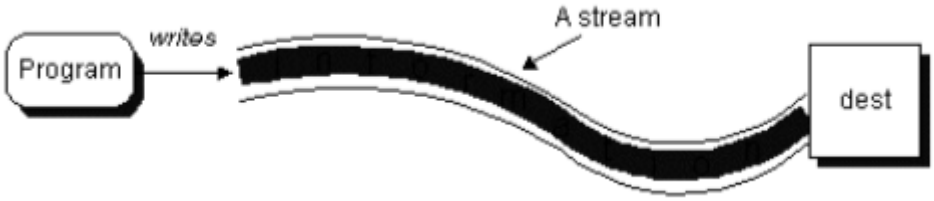
Streams

- Programs receive information from an external source or send out information to an external destination. These two processes are realised with streams. **A stream is an abstraction of a sequence of bytes.**

- To receive information, a program opens a stream on an information source (a file, memory) and reads the information serially, like this:



- A program can send information to an external destination by opening a stream to a destination and writing the information out serially, like this:



- No matter where the information is coming from or going to and no matter what type of data is being read or written, the algorithms for reading and writing data is pretty much always the same.

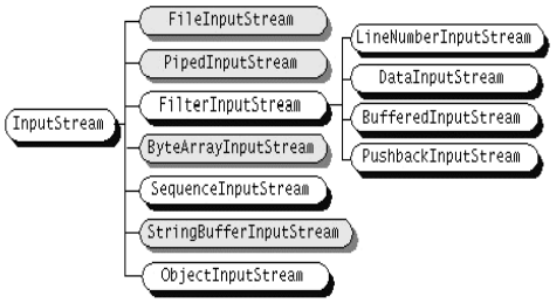
```
Reading
open a stream
while more information
    read information
close the stream
```

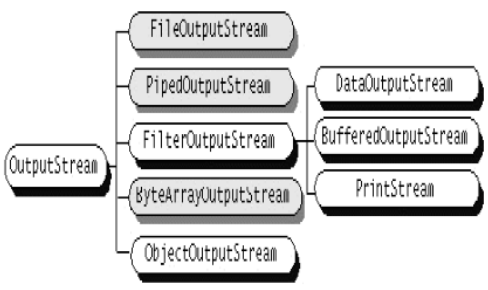
```
Writing
open a stream
while more information
    write information
close the stream
```

- `java.io` contains a set of stream classes that support algorithms for reading and writing. These are divided into 2 classes based on the datae type: **characters** and **bytes**.

Character Streams

- **Reader** and **Writer** are the abstract superclasses for **character streams** in `java.io`.
- **Reader** (Writer) provides the API and partial implementation for readers (writers) -- streams that read (write) 16-bit characters.
- Subclasses of **Reader** and **Writer** implement specialized streams and are divided into two categories: Those that read from or write to data sinks and those that perform some sort of processing.





FileWriter

Constructors

```
FileWriter(File file) // Constructs a FileWriter object given a File object.
FileWriter (File file, boolean append) // constructs a FileWriter object given a File object.
FileWriter (FileDescriptor fd) // constructs a FileWriter object associated with a file descriptor.
FileWriter (String fileName) // constructs a FileWriter object given a file name.
FileWriter (String fileName, Boolean append) // Constructs a FileWriter object given a file name with a Boolean indicating whether or not to append the data written.
```

Methods

```
void write (int c) throws IOException // Writes a single character.
void write (char [] stir) throws IOException // Writes an array of characters.
void write(String str)throws IOException // Writes a string.
void write(String str,int off,int len)throws IOException // Writes a portion of a string. Here off is offset from which to start writing characters and len is number of character to write.
void flush() throws IOException // flushes the stream
void close() throws IOException // flushes the stream first and then closes the writer.
```

Example FileWriter

```
import java.io.FileWriter;
import java.io.IOException;
class CreateFile
{
    public static void main(String[] args) throws IOException
    {
        // Accept a string
        String str = "File Handling in Java using "+
                " FileWriter and FileReader";

        // attach a file to FileWriter
        FileWriter fw=new FileWriter("output.txt");

        // read character wise from string and write
        // into FileWriter
        for (int i = 0; i < str.length(); i++)
            fw.write(str.charAt(i));

        System.out.println("Writing successful");
        //close the file
        fw.close();
    }
}
```

FileReader

- **Definition:** makes it easy to read the contents of a file.

- **Reader** is an abstract base class that makes reading characters possible through one of its concrete implementations.
- It defines the following basic operations of reading characters from any medium such as memory or the filesystem:
  - Read a single character
  - Read an array of characters
  - Mark and reset a given position in a stream of characters
  - Skip position while reading a character stream
  - Close the input stream
- **FileReader** inherits its functionality from **InputStreamReader**, which is a **Reader** implementation designed to read bytes from an input stream as characters.

```
public class InputStreamReader extends Reader {}
public class FileReader extends InputStreamReader {}
```

- You use a **FileReader** when we want to read text from a file using the system's default character set.

Constructors

```
public FileReader(String fileName) throws FileNotFoundException {
    super(new FileInputStream(fileName));
}

public FileReader(File file) throws FileNotFoundException {
    super(new FileInputStream(file));
}

public FileReader(FileDescriptor fd) {
    super(new FileInputStream(fd));
}
```

Read Single Char

```
public static String readAllCharactersOneByOne(Reader reader) throws IOException
{
    StringBuilder content = new StringBuilder();
    int nextChar;
    while ((nextChar = reader.read()) != -1) {
        content.append((char) nextChar);
    }
    return String.valueOf(content);
}

-----------
Main.java
-----------
String expectedText = "Hello, World!";
File file = new File(FILE_PATH);
try (FileReader fileReader = new FileReader(file)) {
    String content = FileReaderExample.readAllCharactersOneByOne(fileReader);
    Assert.assertEquals(expectedText, content);
}
```

Read Array of chars

```
public static String readMultipleCharacters(Reader reader, int length) throws
IOException {
    char[] buffer = new char[length];
    int charactersRead = reader.read(buffer, 0, length);
    if (charactersRead != -1) {
        return new String(buffer, 0, charactersRead);
    } else {
        return "";
    }
}
-----------
Main.java
-----------
String expectedText = "Hello";
File file = new File(FILE_PATH);
try (FileReader fileReader = new FileReader(file)) {
    String content = FileReaderExample.readMultipleCharacters(fileReader, 5);
    Assert.assertEquals(expectedText, content);
}
```

Limitations of FileReader

- Relies on default system character encoding
- Custom values for character set, buffer size or input stream we must use **InputStreamReader**.
- **IO cycle** are expensive and can introduce latency to our app. So it's best to minimize the number of of IO operations by wrapping a **BufferedReader** around our FileReader object.

FileWriter

- **FileWriter** is a specialized **OutputStreamWriter** for writing character files.
- Works with the operations inherited from **OutputStreamWriter** and **Writer** classes.
- Until Java 11, the FileWriter worked with the default character encoding and default byte buffer size. However, Java 11 introduced four new constructors that accept a Charset, thereby allowing user-specified Charset. Unfortunately, we still cannot modify the byte buffer size, and it's set to 8192.

Constructors

```
public FileWriter(String fileName) throws IOException {
    super(new FileOutputStream(fileName));
}

public FileWriter(String fileName, boolean append) throws IOException {
    super(new FileOutputStream(fileName, append));
}

public FileWriter(File file) throws IOException {
    super(new FileOutputStream(file));
}

public FileWriter(File file, boolean append) throws IOException {
    super(new FileOutputStream(file, append));
}
```

```
public FileWriter(FileDescriptor fd) {
    super(new FileOutputStream(fd));
}

// Constructors from Java 11+ with CharSet
public FileWriter(String fileName, Charset charset) throws IOException {
    super(new FileOutputStream(fileName), charset);
}

public FileWriter(String fileName, Charset charset, boolean append) throws
IOException {
    super(new FileOutputStream(fileName, append), charset);
}

public FileWriter(File file, Charset charset) throws IOException {
    super(new FileOutputStream(file), charset);
}

public FileWriter(File file, Charset charset, boolean append) throws IOException
{
    super(new FileOutputStream(file, append), charset);
}
```

Writing a string to a file

```
try (FileWriter fileWriter = new
FileWriter("src/test/resources/FileWriterTest.txt")) {
    fileWriter.write("Hello Folks!");
    // Output: Hello Folks!
}
```

- **Note:** Since the FileWriter is AutoCloseable, we've used try-with-resources so that we don't have to close the FileWriter explicitly.
- The **FileWriter** does **not guarantee** whether the `FileWriterTest.txt` file will be available or be created. It is dependent on the underlying platform.
- We must also make a note that certain platforms may allow only a single FileWriter instance to open the file. In that case, the other constructors of the FileWriter class will fail if the file involved is already open.

Append string to file

```
try (FileWriter fileWriter = new
FileWriter("src/test/resources/FileWriterTest.txt", true)) {
    fileWriter.write("Hello Folks Again!");
}
// Output: Hello Folks!Hello Folks Again!
```

- As we can see, we've used the two-argument constructor that accepts a file name and a boolean flag append.
- Passing the flag **append** as **true** creates a **FileWriter** that allows us to append text to existing contents of a file.

BufferedReader

- BufferedReader comes in handy if we want to read text from any kind of input source whether that be files, sockets, or something else.
- **Simply:** it enables us to minimize the number of I/O operations by reading chunks of characters and storing them in an internal buffer.
- While the buffer has data, the reader will read from it instead of directly from the underlying stream.

Buffering another Reader

- Like most of the Java I/O classes, **BufferedReader** implements **Decorator pattern**, meaning it expects a **Reader** in its **constructor**.
- In this way, it enables us to flexibly extend an instance of a Reader implementation with buffering functionality

```
BufferedReader reader =
  new BufferedReader(new FileReader("src/main/resources/input.txt"));
```

- But, if buffering doesn't matter to us we could just use a FileReader directly:

```
FileReader reader =
  new FileReader("src/main/resources/input.txt");
```

- In addition to buffering, **BufferedReader also provides some nice helper functions for reading files line-by-line**. So, even though it may appear simpler to use FileReader directly, BufferedReader can be a big help.

Buffering a Stream

- we can configure **BufferedReader** to take any kind of **input stream** as an underlying source.
- We can do it using **InputStreamReader** and wrapping it in the constructor:

```
BufferedReader reader =
  new BufferedReader(new InputStreamReader(System.in));
```

- we are reading from System.in which typically corresponds to the input from the keyboard.
- we could pass an input stream for reading from a socket, file or any imaginable type of textual input.
- The only **prerequisite** is that there is a suitable **InputStream** implementation for it.

BufferedReader vs Scanner

- **Scanner** brings same functionality as BufferedReader;
- However, there are significant differences between these two classes which can make them either more or less convenient for us, depending on our use case:
  - **BufferedReader** is synchronized (**thread-safe**) while Scanner is **not**
  - **Scanner** can parse primitive types and strings using regular expressions
  - **BufferedReader** allows for **changing** the size of the buffer while **Scanner** has a **fixed** buffer size
  - **BufferedReader** has a larger default buffer size
  - **Scanner hides** IOException, while **BufferedReader** forces us to **handle** it
  - **BufferedReader** is usually **faster** than **Scanner** because it only reads the data without parsing it

- if we are parsing individual tokens in a file, then **Scanner** will feel a bit more natural than **BufferedReader**. But, just reading a line at a time is where **BufferedReader** shines.

Reading Text with Buf.Read

Initializing

```
BufferedReader reader =
  new BufferedReader(new FileReader("src/main/resources/input.txt"));

// sets buffer size to 16384 bytes (16kb)
BufferedReader reader =
  new BufferedReader(new FileReader("src/main/resources/input.txt")), 16384);
```

Read Line-by-line

```
public String readAllLines(BufferedReader reader) throws IOException {
    StringBuilder content = new StringBuilder();
    String line;

    while ((line = reader.readLine()) != null) {
        content.append(line);
        content.append(System.lineSeparator());
    }

    return content.toString();
}

// OR from java8+

public String readAllLinesWithStream(BufferedReader reader) {
    return reader.lines()
      .collect(Collectors.joining(System.lineSeparator()));
}
```

Closing the stream

```
try (BufferedReader reader =
       new BufferedReader(new FileReader("src/main/resources/input.txt"))) {
    return readAllLines(reader);
}
```

Read single char

```
public String readAllCharsOneByOne(BufferedReader reader) throws IOException {
    StringBuilder content = new StringBuilder();

    int value;
    while ((value = reader.read()) != -1) {
        content.append((char) value);
    }

    return content.toString();
}
```

Read multiple char

```
public String readMultipleChars(BufferedReader reader) throws IOException {
    int length;
    char[] chars = new char[length];
    int charsRead = reader.read(chars, 0, length);

    String result;
    if (charsRead != -1) {
        result = new String(chars, 0, charsRead);
    } else {
        result = "";
    }

    return result;
}
```

Skipping char

```
public void givenBufferedReader_whensSkipChars_thenOk() throws IOException {
    StringBuilder result = new StringBuilder();

    try (BufferedReader reader =
           new BufferedReader(new StringReader("1__2__3__4__5"))) {
        int value;
        while ((value = reader.read()) != -1) {
            result.append((char) value);
            reader.skip(2L);
        }
    }

    assertEquals("12345", result);
}
```

Mark & Reset

```
public void givenBufferedReader_whenSkipsWhitespacesAtBeginning_thenOk()
  throws IOException {
    String result;

    try (BufferedReader reader =
           new BufferedReader(new StringReader("    Lorem ipsum dolor sit
amet."))) {
        do {
            reader.mark(1);
        } while(Character.isWhitespace(reader.read()))

        reader.reset();
        result = reader.readLine();
    }

    assertEquals("Lorem ipsum dolor sit amet.", result);
}
```

Example PrintWriter

```
import java.io.*;
```

```java
import java.util.Locale;
//Java program to demonstrate PrintWriter
class PrintWriterDemo {

    public static void main(String[] args)
        {
        String s="GeeksforGeeks";

        // create a new writer
        PrintWriter out = new PrintWriter(System.out);
        char c[]={'G','E','E','K'};

        //illustrating print(boolean b) method
        out.print(true);

        //illustrating print(int i) method
        out.print(1);

        //illustrating print(float f) method
        out.print(4.533f);

        //illustrating print(String s) method
        out.print("GeeksforGeeks");
        out.println();

        //illustrating print(Object Obj) method
        out.print(out);
        out.println();

        //illustrating append(CharSequence csq) method
        out.append("Geek");
        out.println();

        //illustrating checkError() method
        out.println(out.checkError());

        //illustrating format() method
        out.format(Locale.UK, "This is my %s program", s);

        //illustrating flush method
        out.flush();

        //illustrating close method
        out.close();
    }
}
```

Output:

```
true14.533GeeksforGeeks
        java.io.PrintWriter@1540e19d
        Geek
        false
        This is my GeeksforGeeks program
```

```
PrintWriterout = new PrintWriter("output.txt"); out.println(29.95);
out.println(new Rectangle(5,10,15,25));out.print("Hello, World!");out.close();
```

```
public class Product
{
        public boolean read(Scanner in) throws IOException {
                if(!in.hasNext()) return false;
                name = in.nextLine();
                if (name == null) return false;
                if (!in.hasNext()) throw new EOFException("Missing price");
                String inputLine = in.nextLine();
                price = Double.parseDouble(inputLine);
                if (!in.hasNext()) throw new EOFException("Missing score");
                inputLine = in.nextLine();
                score = Integer.parseInt(inputLine);
                return true;
        }
        public String name;
        private double price;
        private int score;
}
 public ArrayList<Product> readProducts(Scanner in) throws IOException
 {
        result = new ArrayList<Product>();
        boolean done = false;
        while (!done) {
                Product p = new Product();
                if (p.read(in))
                        result.add(p);
                else done = true;
        }
        return result;
 }
```

```
public void openProcessFile() {
    ArrayList<Product> result = new ArrayList<Product> ();
    try(Scanner in = new Scanner(new FileReader(fileName));
    { // select file name
        result = readProducts(in);
        ...
    }

    catch(FileNotFoundException e)
    { System.out.println("Bad file name");}
    catch(IOException e)
    { System.out.println("Corrupted file");}
}
```

RandomAccessFile

Sequential access

Random access

file pointer

Random access in Java

file pointer

```
<fileVariable>.writeChars(<String>); //<strLength>*2 bytes
<fileVariable>.writeInt(<int>);      // 4 bytes
<fileVariable>.writeDouble(<double>);// 8 bytes
```

file pointer of

```
<fileVariable>.readChar();     // 2 bytes
<fileVariable>.readInt();      // 4 bytes
<fileVariable>.readDouble();   // 8 bytes
```

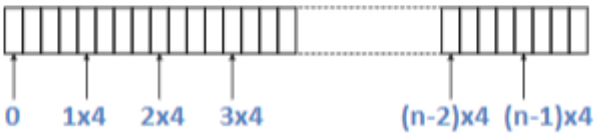
All RandomAccessFile Methods throw exceptions (IO, EOF etc.) !!!

file pointer of
`<fileVariable>`

1. Integers

Logical format

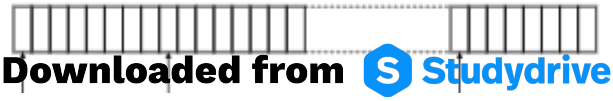
Physical format



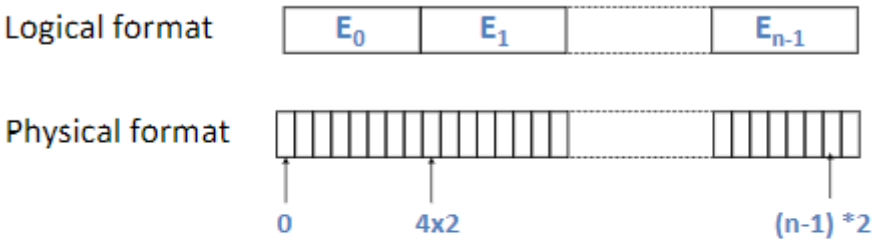
2. Doubles

Logical format

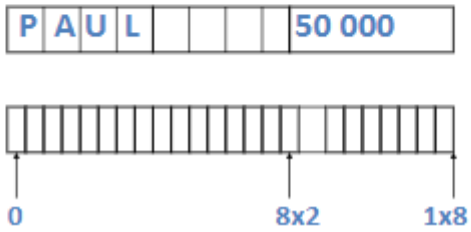
Physical format



3. Strings



4. Mixed Data



Constructors

```
RandomAccessFile(File f, String operation)
```

Methods

```
read() // java.io.RandomAccessFile.read() reads byte of data from file. The byte is returned as an integer in the range 0-255
read(byte[] b) // reads bytes upto b.length from the buffer.
read((byte[] b, int offset, int len) // reads bytes initialising from offset position upto b.length from the buffer.
readBoolean() // reads a boolean from the file.
readByte() // reads a signed eight-bit value from file, start reading from the File Pointer.
readChar() :// reads a character from the file, start reading from the File Pointer.
readDouble() // reads a double value from the file, start reading from the File Pointer.
readFloat() // reads a float value from the file, start reading from the File Pointer.
readFully(byte[] b) // reads bytes upto b.length from the buffer, start reading from the File Pointer.
readInt() // reads a signed 4 bytes integer from the file, start reading from the File Pointer.
readFully(byte[] b, int offset, int len) // reads bytes initialising from offset position upto b.length from the buffer, start reading from the File Pointer.
readLong() // reads a signed 64 bit integer from the file, start reading from the File Pointer.
```

Example of RandomAccessFile

```
import java.io.*;
public class NewClass
{
    public static void main(String[] args)
    {
        try
```

```
    {
        double d = 1.5;
        float f = 14.56f;

        // Creating a new RandomAccessFile - "GEEK"
        RandomAccessFile geek = new RandomAccessFile("GEEK.txt", "rw");

        // Writing to file
        geek.writeUTF("Hello Geeks For Geeks");

        // File Pointer at index position - 0
        geek.seek(0);

        // read() method :
        System.out.println("Use of read() method : " + geek.read());

        geek.seek(0);

        byte[] b = {1, 2, 3};

        // Use of .read(byte[] b) method :
        System.out.println("Use of .read(byte[] b) : " + geek.read(b));

        // readBoolean() method :
        System.out.println("Use of readBoolean() : " + geek.readBoolean());

        // readByte() method :
        System.out.println("Use of readByte() : " + geek.readByte());

        geek.writeChar('c');
        geek.seek(0);

        // readChar() :
        System.out.println("Use of readChar() : " + geek.readChar());

        geek.seek(0);
        geek.writeDouble(d);
        geek.seek(0);

        // read double
        System.out.println("Use of readDouble() : " + geek.readDouble());

        geek.seek(0);
        geek.writeFloat(f);
        geek.seek(0);

        // readFloat() :
        System.out.println("Use of readFloat() : " + geek.readFloat());

        geek.seek(0);
        // Create array upto geek.length
        byte[] arr = new byte[(int) geek.length()];
        // readFully() :
        geek.readFully(arr);

        String str1 = new String(arr);
        System.out.println("Use of readFully() : " + str1);
```

```
            geek.seek(0);

            // readFully(byte[] b, int off, int len) :
            geek.readFully(arr, 0, 8);

            String str2 = new String(arr);
            System.out.println("Use of readFully(byte[] b, int off, int len) : "
+ str2);
        }
        catch (IOException ex)
        {
            System.out.println("Something went Wrong");
            ex.printStackTrace();
        }
    }
}
```

Output:

```
Use of read() method : 0
Use of .read(byte[] b) : 3
Use of readBoolean() : true
Use of readByte() : 108
Use of readChar() : c
Use of readDouble() : 1.5
Use of readFloat() : 14.56
Use of readFully() : Geeks For Geeks
Use of readFully(byte[] b, int off, int len) : Geeks For Geeks
```

```
private void writeToPosition(String filename, int data, long position)
  throws IOException {
    RandomAccessFile writer = new RandomAccessFile(filename, "rw");
    writer.seek(position);
    writer.writeInt(data);
    writer.close();
}

private int readFromPosition(String filename, long position)
  throws IOException {
    int result = 0;
    RandomAccessFile reader = new RandomAccessFile(filename, "r");
    reader.seek(position);
    result = reader.readInt();
    reader.close();
    return result;
}

@Test
public void whenWritingToSpecificPositionInFile_thenCorrect()
  throws IOException {
    int data1 = 2014;
    int data2 = 1500;

    writeToPosition(fileName, data1, 4);
    assertEquals(data1, readFromPosition(fileName, 4));
```

```
    writeToPosition(fileName2, data2, 4);
    assertEquals(data2, readFromPosition(fileName, 4));
}
```

Object Streams

- The class **ObjectOutputStream** can save entire objects in files.
- The class **ObjectInputStream** can read entire objects from files

```
Employeee = new Employee("John", 20000);
ObjectOutputStreamout = new ObjectOutputStream(new FileOutputStream("e.dat"));
out.writeObject(e);
```

Example of Object Streams

```
public class Person implements Serializable {

    private static final long serialVersionUID = 1L;
    private String name;
    private int age;
    private String gender;

    Person() {
    };

    Person(String name, int age, String gender) {
        this.name = name;
        this.age = age;
        this.gender = gender;
    }

    @Override
    public String toString() {
        return "Name:" + name + "\nAge: " + age + "\nGender: " + gender;
    }
}
```

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class WriterReader {

    public static void main(String[] args) {

        Person p1 = new Person("John", 30, "Male");
        Person p2 = new Person("Rachel", 25, "Female");

        try {
            FileOutputStream f = new FileOutputStream(new
File("myObjects.txt"));
            ObjectOutputStream o = new ObjectOutputStream(f);
```

```
            // Write objects to file
            o.writeObject(p1);
            o.writeObject(p2);

            o.close();
            f.close();

            FileInputStream fi = new FileInputStream(new File("myObjects.txt"));
            ObjectInputStream oi = new ObjectInputStream(fi);

            // Read objects
            Person pr1 = (Person) oi.readObject();
            Person pr2 = (Person) oi.readObject();

            System.out.println(pr1.toString());
            System.out.println(pr2.toString());

            oi.close();
            fi.close();

        } catch (FileNotFoundException e) {
            System.out.println("File not found");
        } catch (IOException e) {
            System.out.println("Error initializing stream");
        } catch (ClassNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }

}
```

Output:

```
Name:John
Age: 30
Gender: Male
Name:Rachel
Age: 25
Gender: Female
```

Read & Write Array of Objects

```
public class ExerciseSerializableNew {

    public static void main(String...args) {
        //create a Serializable List
        List < String > quarks = Arrays.asList(
            "up", "down", "strange", "charm", "top", "bottom"
        );

        //serialize the List
        try (
```

```
            OutputStream file = new FileOutputStream("quarks.ser"); OutputStream
buffer = new BufferedOutputStream(file); ObjectOutput output = new
ObjectOutputStream(buffer);
        ) {
            output.writeObject(quarks);
        } catch (IOException ex) {
            logger.log(Level.SEVERE, "Cannot perform output.", ex);
        }

        //deserialize the quarks.ser file
        try (
            InputStream file = new FileInputStream("quarks.ser"); InputStream
buffer = new BufferedInputStream(file); ObjectInput input = new
ObjectInputStream(buffer);
        ) {
            //deserialize the List
            List < String > recoveredQuarks = (List < String > )
input.readObject();
            //display its data
            for (String quark: recoveredQuarks) {
                System.out.println("Recovered Quark: " + quark);
            }
        } catch (ClassNotFoundException ex) {
            logger.log(Level.SEVERE, "Cannot perform input. Class not found.",
ex);
        } catch (IOException ex) {
            logger.log(Level.SEVERE, "Cannot perform input.", ex);
        }
    }

    // PRIVATE

    private static final Logger logger =
        Logger.getLogger(ExerciseSerializableNew.class.getPackage().getName());
}
```

FileOutputStream

```
@Test
public void givenWritingStringToFile_whenUsingFileOutputStream_thenCorrect()
  throws IOException {
    String str = "Hello";
    FileOutputStream outputStream = new FileOutputStream(fileName);
    byte[] strToBytes = str.getBytes();
    outputStream.write(strToBytes);

    outputStream.close();
}
```

DataOutputStream

```
@Test
public void givenWritingToFile_whenUsingDataOutputStream_thenCorrect()
  throws IOException {
    String value = "Hello";
    FileOutputStream fos = new FileOutputStream(fileName);
```

```
    DataOutputStream outStream = new DataOutputStream(new
BufferedOutputStream(fos));
    outStream.writeUTF(value);
    outStream.close();

    // verify the results
    String result;
    FileInputStream fis = new FileInputStream(fileName);
    DataInputStream reader = new DataInputStream(fis);
    result = reader.readUTF();
    reader.close();

    assertEquals(value, result);
}
```

DataInputStream

```
@Test
public void whenReadWithDataInputStream_thenCorrect() throws IOException {
    String expectedValue = "Hello, world!";
    String file ="src/test/resources/fileTest.txt";
    String result = null;

    DataInputStream reader = new DataInputStream(new FileInputStream(file));
    int nBytesToRead = reader.available();
    if(nBytesToRead > 0) {
        byte[] bytes = new byte[nBytesToRead];
        reader.read(bytes);
        result = new String(bytes);
    }

    assertEquals(expectedValue, result);
}
```

FileClass

- File class describes disk files and directories
- Create a File object

```
File file = new File("file.txt");
```

- Some file methods: delete, renameTo, exists

JFileChooserDialog

Methods:

- Construct a file chooser object: `JFileChooser`
- To create dialog window: `showOpenDialog` or `showSaveDialog`

  (You can specify null or the user interface component over which to pop up the dialog)
- If the user chooses a file, these methods return:
  `JFileChooser.APPROVE_OPTION`
- If the user cancels the selection, these methods return:
  `JFileChooser.CANCEL_OPTION`
- If a file is chosen, you can use `GetSelectedFile` method to obtain a `File` object describing the file.

```
JFileChooser chooser = new JFileChooser();
FileReader in = null;
if ( chooser.showOpenDialog(null) ==
        JFileChooser.APPROVE_OPTION )
{
    File selectedFile = chooser.getSelectedFile();
    in = new FileReader(selectedFile);
}
```

```
public static void main(String[] args) {
    try {
        FileWriterwriter = new FileWriter(args[0]);
        PrintWriterout = new PrintWriter(writer);
        out.println(29.95);
        out.println(39.05);
        writer.close();
        FileReaderreader = new FileReader(args[0]);
        BufferedReaderin = new BufferedReader(reader);
        StringinputLine = in .readLine();
        double x = Double.parseDouble(inputLine);
        System.out.println(x);
        inputLine = in .readLine();
        x = Double.parseDouble(inputLine);
        System.out.println(x);
        reader.close();
    } catch (Throwablee) {
        System.out.println("ourError");
    }
} // to run the program type "java readWriteio.txt"
```