Collections and Generics

Mastek

MASTEK INSTITUTE
FOR LEARNING & DEVELOPMENT

Problems with Arrays

a. Fixed Length
b. Managing Unique values
c. Using only Integer as index
d. Memory Management limitations
e. Data Operations on array requires additional logic

Mastek

Abstract Data Types

Data Structures and Algorithms

Standard

Well-understood

Efficient

Examples

- Stack, queue, linked list

Mastek

Interface-based design

Separate interface from implementation

Built in to Java language

Polymorphism

- List l = new LinkedList();
- Calling l.add() invokes method of class LinkedList

Mastek

Collections Framework

Interoperability between unrelated APIs

Reduces the effort required to learn APIs

Reduces the effort required to design and implement APIs

Fosters software reuse

Mastek

Overview: Core Interfaces

Collection

Set

List

Map

Mastek

Collection

A group of objects

Major methods:

- int size();
- boolean isEmpty();
- boolean contains(Object);
- Iterator iterator();
- Object[] toArray();
- boolean add(Object);
- boolean remove(Object);
- void clear();

Mastek

List

interface List extends Collection

An ordered collection of objects

Duplicates allowed

Implemented by:

- ArrayList, LinkedList, Vector

Mastek

List Implementations

ArrayList

- a resizable-array implementation like Vector
- unsynchronized, and without legacy methods

LinkedList

- a doubly-linked list implementation
- May provide better performance than ArrayList
- if elements frequently inserted/deleted within the List
- For queues and double-ended queues (deques)

Vector

- a synchronized resizable-array implementation of a List with additional "legacy" methods.

Mastek

List Details

Major additional methods:

- E get(int);
- E set(int, E);
- int indexOf(E);
- int lastIndexOf(E);
- void add(int, E);
- E remove(int);
- List subList(int, int);
- add() inserts
- remove() deletes

Mastek

Set

interface Set extends Collection

An unordered collection of objects

No duplicate elements

Same methods as Collection

- Semantics are different, so different interface needed for design

Implemented by:

- HashSet, TreeSet

Mastek

Set Implementations

HashSet

- a Set backed by a hash table

TreeSet

- A balanced binary tree implementation
- Imposes an ordering on its elements

Mastek

Map

interface Map (does not extend Collection)

An object that maps keys to values

Each key can have at most one value

Ordering may be provided by implementation class, but not guaranteed

Mastek

Map Details

Major methods:

- int size();
- boolean isEmpty();
- boolean containsKey(E);
- boolean containsValue(E);
- V get(K);
- V put(K, V);
- V remove(K);
- void putAll(Map);
- void clear();

Implemented by:

- HashMap, Hashtable, WeakHashMap, Attributes

Mastek

Accessing all members of Map

Methods

- Set keySet();
- Collection values();
- Set entrySet();

Map.Entry

- Object that contains a key-value pair
- getKey(), getValue()

The collections returned are backed by the map

- When the map changes, the collection changes

Behavior can easily become undefined

- Be very careful and read the docs closely

Mastek

Map Implementations

HashMap

- A hash table implementation of Map
- Like Hashtable, but supports null keys & values

TreeMap

- A balanced binary tree implementation
- Imposes an ordering on its elements

Hashtable

- Synchronized hash table implementation of Map interface, with additional "legacy" methods.

Mastek

Generics

Mastek™

What is Generics?

Generics provides abstraction over Types

Generics makes type safe code possible

Generics provides increased readability

Mastek

Why Generics?

Problem: Collection element types

- Compiler is unable to verify types
- Assignment must have type casting
- ClassCastException can occur during runtime

Solution: Generics

- Tell the compiler type of the collection
- Let the compiler fill in the cast
- Example: Compiler will check if you are adding Integer type entry to a String type collection

Mastek

Generics

Definitions: LinkedList<E> has a type parameter E that represents the type of the elements stored in the list

Usage: Replace type parameter <E> with concrete type argument, like <Integer> or <MyType>

LinkedList<Integer> can store only Integer or sub-type of Integer as elements

- LinkedList<Integer> li = new LinkedList<Integer>();
- li.add(new Integer(0));
- Integer i = li.iterator().next();

Mastek

Usage of Generics

Instantiate a generic class to create type specific object

All collection classes are rewritten to be generic classes

Generic class can have multiple type parameters

Type argument can be a custom type

- Vector<String> vs = new Vector<String>();
- HashMap<String, Mammal> map =
- new HashMap<String, Mammal>();

Mastek

Generics and Sub-typing

Why this compile error? It is because if it is allowed, ClassCastException can occur during runtime – this is not type-safe

- ArrayList<Integer> ai = new ArrayList<Integer>();
- ArrayList<Object> ao = ai; // If it is allowed at compile time,
- ao.add(new Object());
- Integer i = ai.get(0); // This would result in runtime ClassCastException

So there is no inheritance relationship between type arguments of a generic class

Mastek

Generics and Sub-typing

The following code work

- ArrayList<Integer> ai = new ArrayList<Integer>();
- List<Integer> li = new ArrayList<Integer>();
- Collection<Integer> ci = new ArrayList<Integer>();
- Collection<String> cs = new Vector<String>(4);

Inheritance relationship between Generic classes themselves still exist

Mastek

Generics and Sub-typing

The following code work

- ArrayList<Number> an = new
- ArrayList<Number>();
- an.add(new Integer(5));
- an.add(new Long(1000L));
- an.add(new String(“hello”)); // compile error

Entries in a collection maintain inheritance relationship

Mastek

Generics and Type Erasure

Generic type instantiated with no type arguments is known as Raw Type

Pre-J2SE 5.0 classes continue to function over J2SE 5.0 JVM as raw type

- // Generic type instantiated with type argument
- List<String> ls = new LinkedList<String>();
- // Generic type instantiated with no type
- // argument – This is Raw type
- List lraw = new LinkedList();

Mastek

Type Safe code

The compiler guarantees that either:

- the code it generates will be type-correct at run time, or
- it will output a warning (using Raw type) at compile time

If your code compiles without warnings and has no casts, then you will never get a ClassCastException

- This is “type safe” code

Mastek

Unit Testing using Junit

Mastek™

Overview of Unit Testing Concepts

A unit is the smallest testable part of an application. In Java, and other object-oriented languages, a unit is a method.

Testing is a way of evaluating software, to determine if requirements and expectations are met, and to detect errors.

Mastek

Overview of Unit Testing Concepts

Scope of Unit Testing

For newly generated/developed code

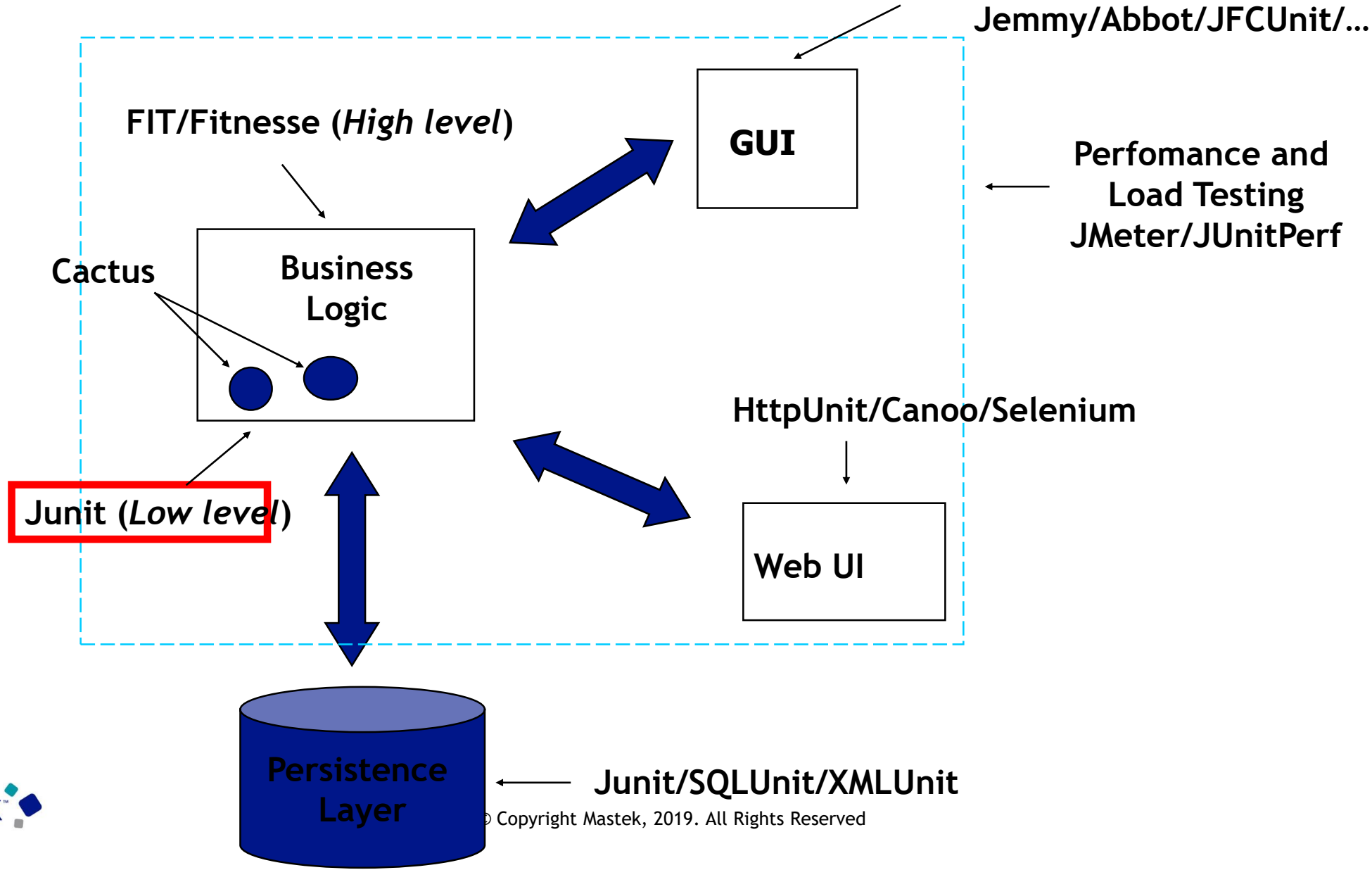
- All units/components of the code

For changed/modified code

- All the affected units/components of the code along with the units/components that were directly changed

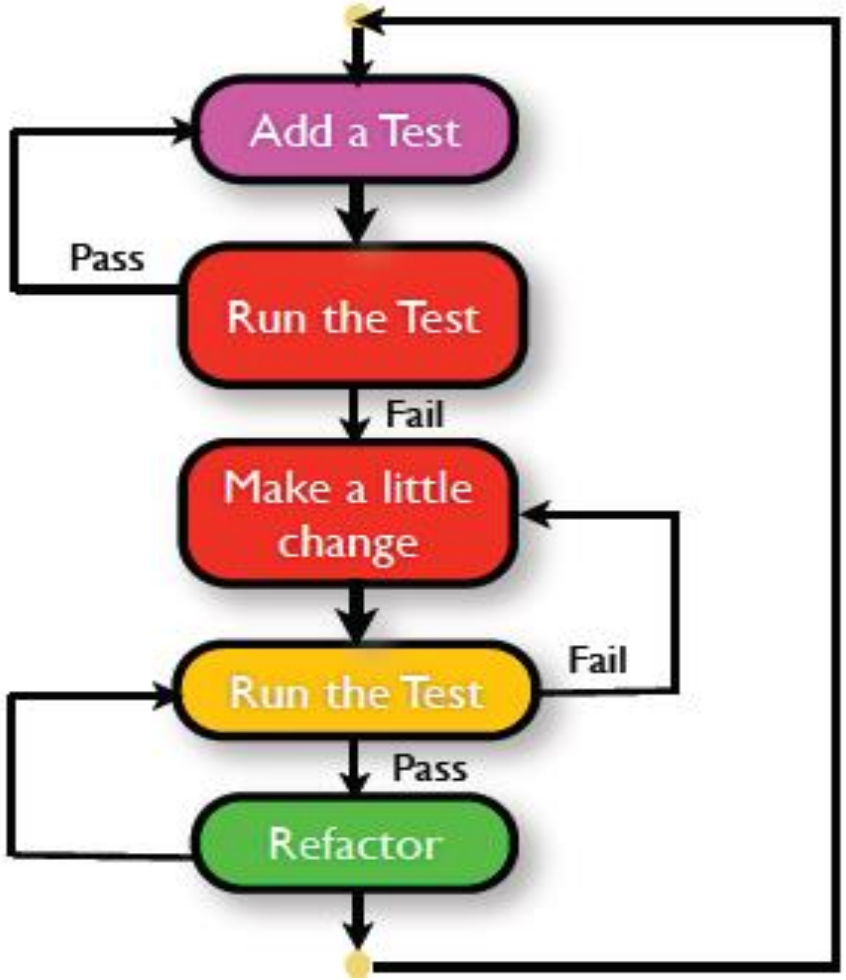
Mastek

Testing Tools



Mastek

What is TDD: Test, Code and Refactor



Mastek

Test Driven Development

- An iterative technique to develop software
- One must first write a test that fails before s/he writes a new functional code.
- A practice for efficiently evolving useful code
- Use TDD to produce the simplest thing that works (but not the dumbest!) [KISS]
- Drive the design of the software through tests
- Focus on writing simple solutions for today's requirements [YAGNI]
- Write just enough code to make the tests pass, and no more
- Executable code (tests) becomes your requirement

Mastek

What is Junit

Is a unit test framework in java

Developed by Kent Beck and Erich Gamma

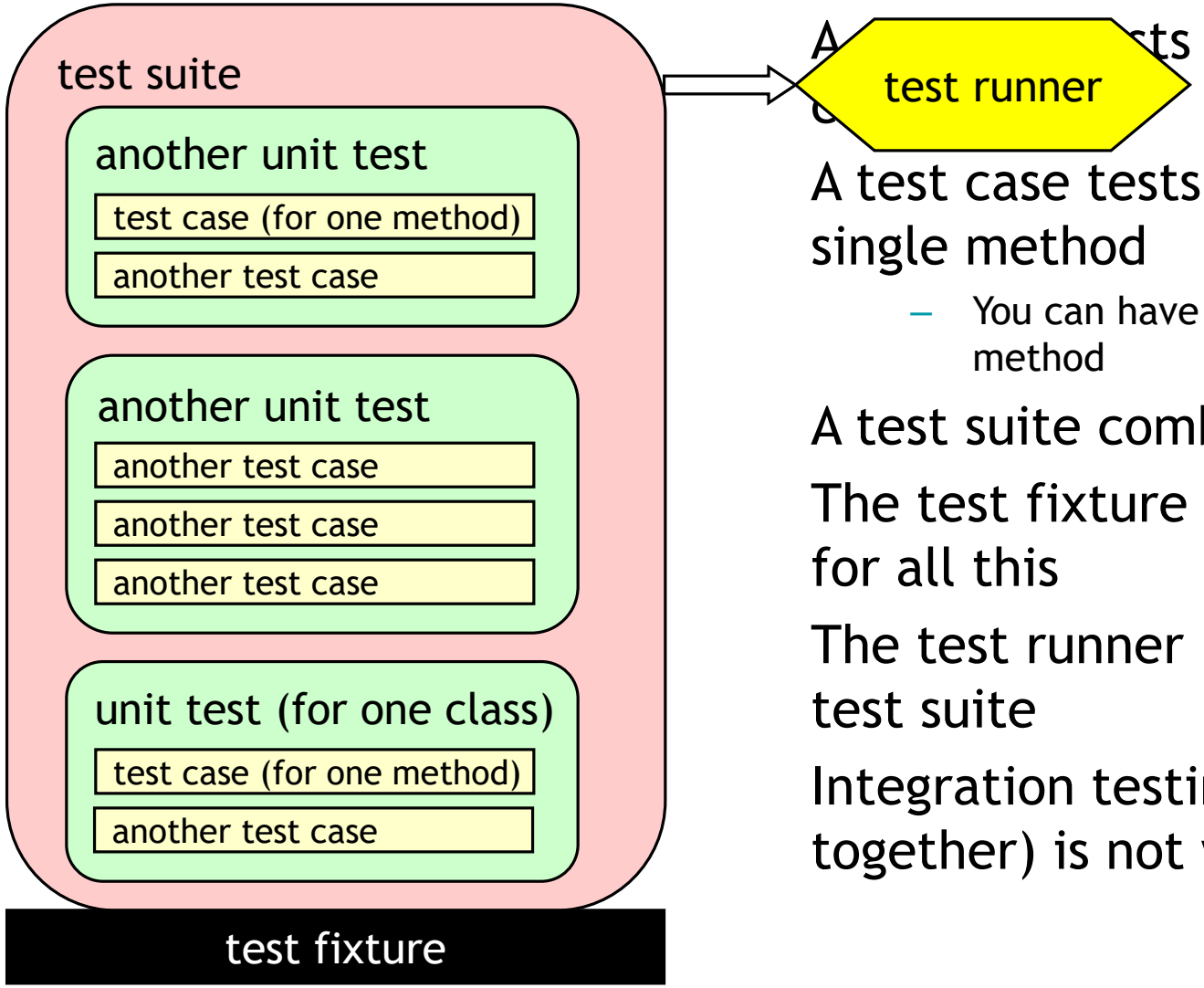
Widely used and commonly become standard unit test framework

Is part of xUnit family. xUnit is a ported Junit for various language.

- PHPunit (PHP)
- Nunit(.NET)

Mastek

Unit Testing concepts, in pictures



A test runner tests the methods in a single class

A test case tests (insofar as possible) a single method

- You can have multiple test cases for a single method

A test suite combines unit tests

The test fixture provides software support for all this

The test runner runs unit tests or an entire test suite

Integration testing (testing that it all works together) is not well supported by JUnit

Mastek

Writing a JUnit test class, I

Start by importing the JUnit 4 classes you need

```
import org.junit.*;
import static org.junit.Assert.*;
```

- Declare your class in the usual way

```
public class MyProgramTest {
```

- Declare any variables you are going to use frequently, typically including an instance of the class being tested

```
MyProgram program;
int [] array;
int solution;
```

Mastek

Writing a JUnit test class, II

- If you wish, you can declare one method to be executed just once, when the class is first loaded
- This is for expensive setup, such as connecting to a database

```
@BeforeClass
public static void setUpClass() throws Exception {
        // one-time initialization code
}
```

- If you wish, you can declare one method to be executed just once, to do cleanup after all the tests have been completed

```
@AfterClass
public static void tearDownClass() throws Exception {
        // one-time cleanup code
}
```

Mastek

Writing a JUnit test class, III

- You can define one or more methods to be executed before each test; typically such methods initialize values, so that each test starts with a fresh set

```
@Before
public void setUp() {
    program = new MyProgram();
    array = new int[] { 1, 2, 3, 4, 5 };
}
```

- You can define one or more methods to be executed after each test; typically such methods release resources, such as files

```
@After
public void tearDown() {
}
```

Mastek

@Before and @After methods

You can have as many @Before and @After methods as you want

- Be warned: You don't know in what order they will execute

You can inherit @Before and @After methods from a superclass; execution is as follows:

- Execute the @Before methods in the superclass
- Execute the @Before methods in this class
- Execute a @Test method in this class
- Execute the @After methods in this class
- Execute the @After methods in the superclass

Mastek

JUnit assertion methods

`assertTrue(`**test**`)`	fails if the boolean test is `false`
`assertFalse(`**test**`)`	fails if the boolean test is `true`
`assertEquals(`**expected**`, `**actual**`)`	fails if the values are not equal
`assertSame(`**expected**`, `**actual**`)`	fails if the values are not the same (by ==)
`assertNotSame(`**expected**`, `**actual**`)`	fails if the values *are* the same (by ==)
`assertNull(`**value**`)`	fails if the given value is *not* `null`
`assertNotNull(`**value**`)`	fails if the given value is `null`
`fail()`	causes current test to immediately fail

Each method can also be passed a string to display if it fails:

- e.g. assertEquals("message", expected, actual)
- Why is there no pass method?

Mastek

Writing a JUnit test class, IV

- A test method is annotated with @Test, takes no parameters, and returns no result
- All the usual assertXXX methods can be used

```
@Test
public void sum() {
    assertEquals(15, program.sum(array));
    assertTrue(program.min(array) > 0);
}
```

Mastek

Special features of @Test

- You can limit how long a method is allowed to take
- This is good protection against infinite loops
- The time limit is specified in milliseconds
- The test fails if the method takes too long

```
@Test (timeout=10)
 public void greatBig() {
    assertTrue(program.ackerman(5, 5) > 10e12);
}
```

- Some method calls should throw an exception
- You can specify that a particular exception is expected
- The test will pass if the expected exception is thrown, and fail otherwise

```
@Test (expected=IllegalArgumentException.class)
public void factorial() {
    program.factorial(-5);
}
```

Mastek

Real Life Unit Testing: Mykart

Add Item in Cart

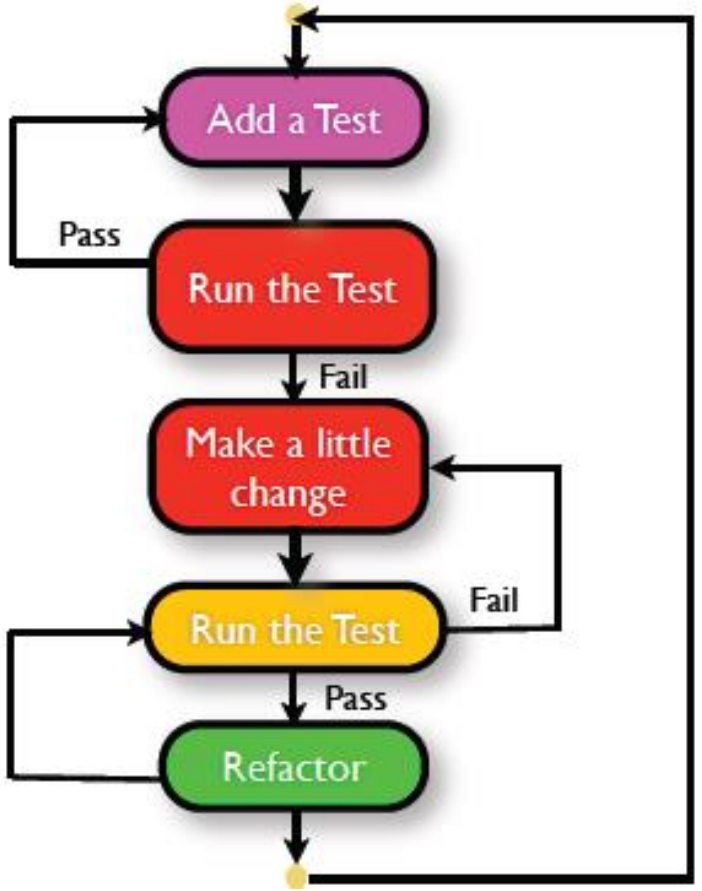
Get total price of Cart

Exception for negative Price, Quantity

Add VAT component

Print the invoice for a Cart

- Item Name
- Quantity
- Unit Price
- Item Total
- Sub Total
- Service Tax
- Grand Total



Mastek

I/O Streams

Mastek™

Files

The file system is maintained by the operating system.

The system provides commands and/or GUI utilities for viewing file directories and for copying, moving, renaming, and deleting files.

The system also provides “core” functions, callable from programs, for reading and writing directories and files.

Mastek

public class File

Information about files, not their contents

Constructors

File(String path) or (String path, String name) or
(File dir, String name)

Methods

boolean exists(), isFile(), isDirectory(),
canRead(), canWrite();

long length(), lastModified();

boolean delete(), mkdir(), mkdirs(),
renameTo(File dest);

String getName(), getParent(), getPath(),
getAbsolutePath()

Mastek

Streams

What is a stream?

Byte Streams and Character Streams

The Predefined Streams

Mastek

What is a stream?

Java programs perform I/O through stream.

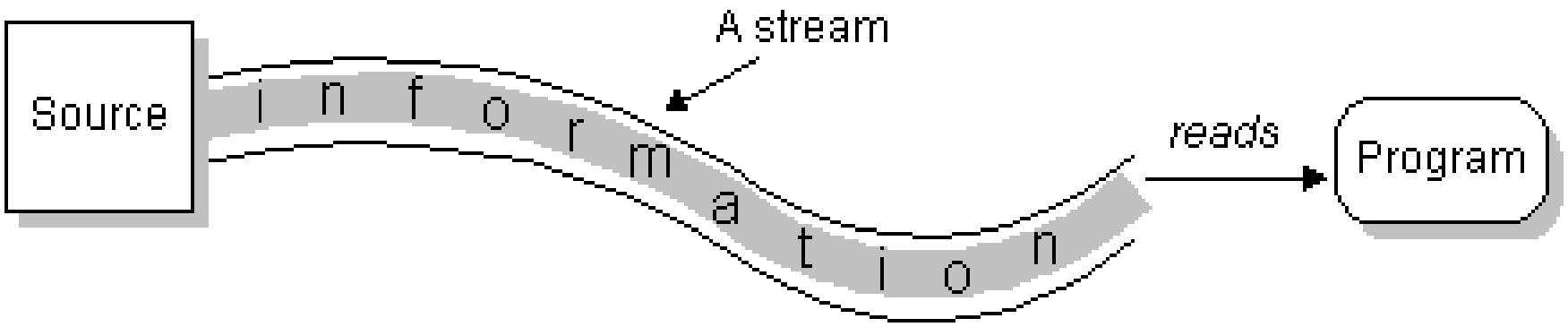
A stream is an abstraction that either produces or consumes information.

A stream is linked to a physical device by the Java I/O system.

Java implements streams within class hierarchies defined in java.io package.

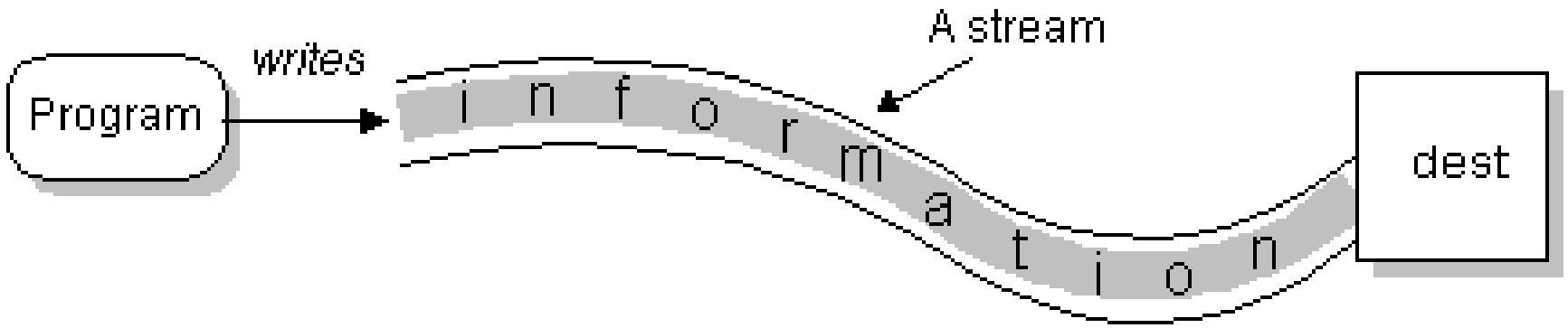
Mastek

Reading from a stream



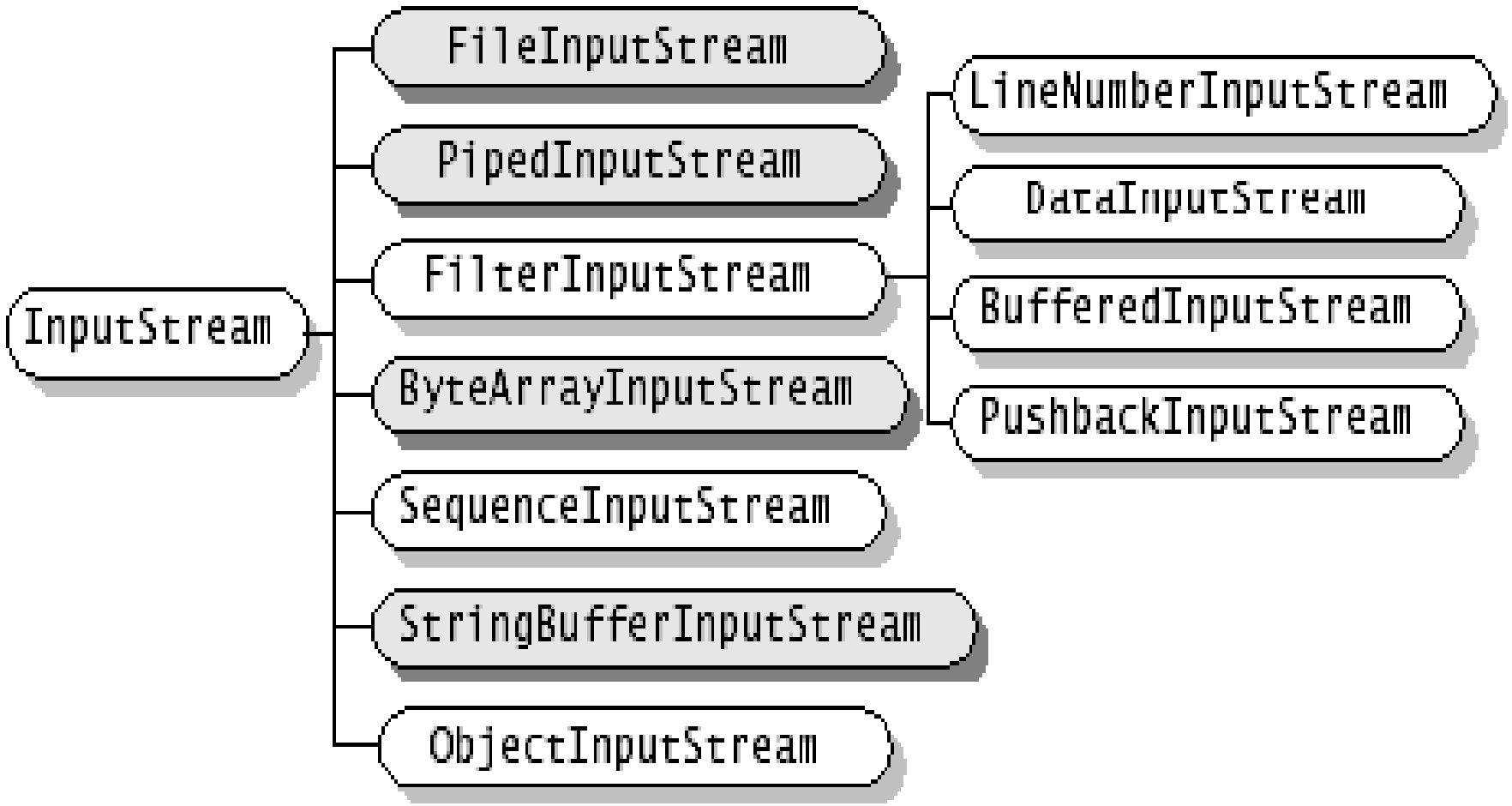
Mastek

Writing to a stream



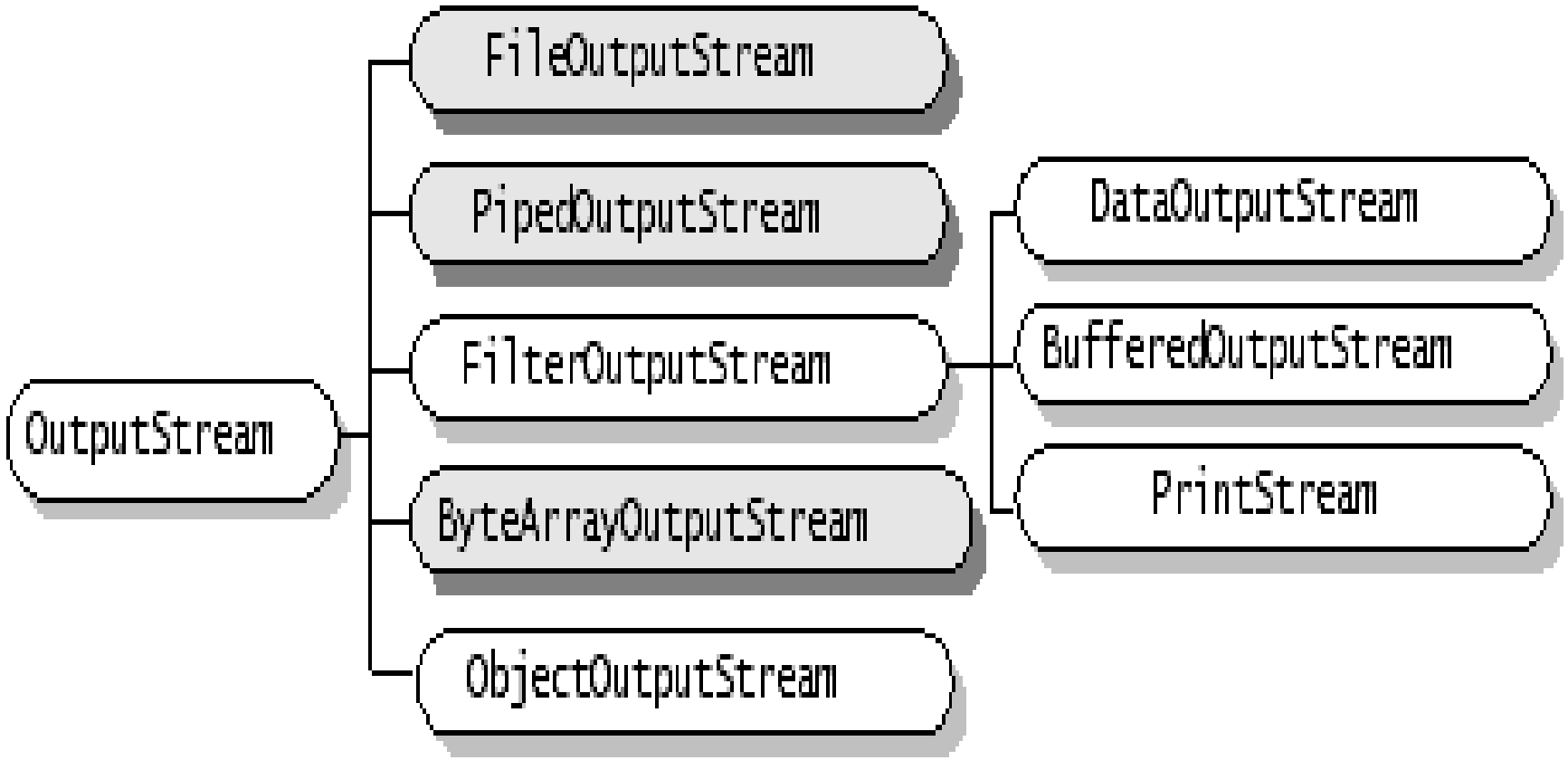
Mastek

The Byte Stream classes



Mastek

The Byte Stream classes



Mastek

Reading and writing files

FileInputStream(String fileName) throws FileNotFoundException

FileOutputStream(String fileName) throws FileNotFoundException

void close() throws IOException

int read() throws IOException

void write(byte[] byteval) throws IOException

Mastek

Writing Data

DataOutputStream

- writeInt(int)
- writeDouble(double)
- wirteUTF(String)

DataInputStream

- readInt()
- readDouble()
- readUTF

Mastek

Writing Objects

ObjectOutputStream

- writeObject(Serializable)

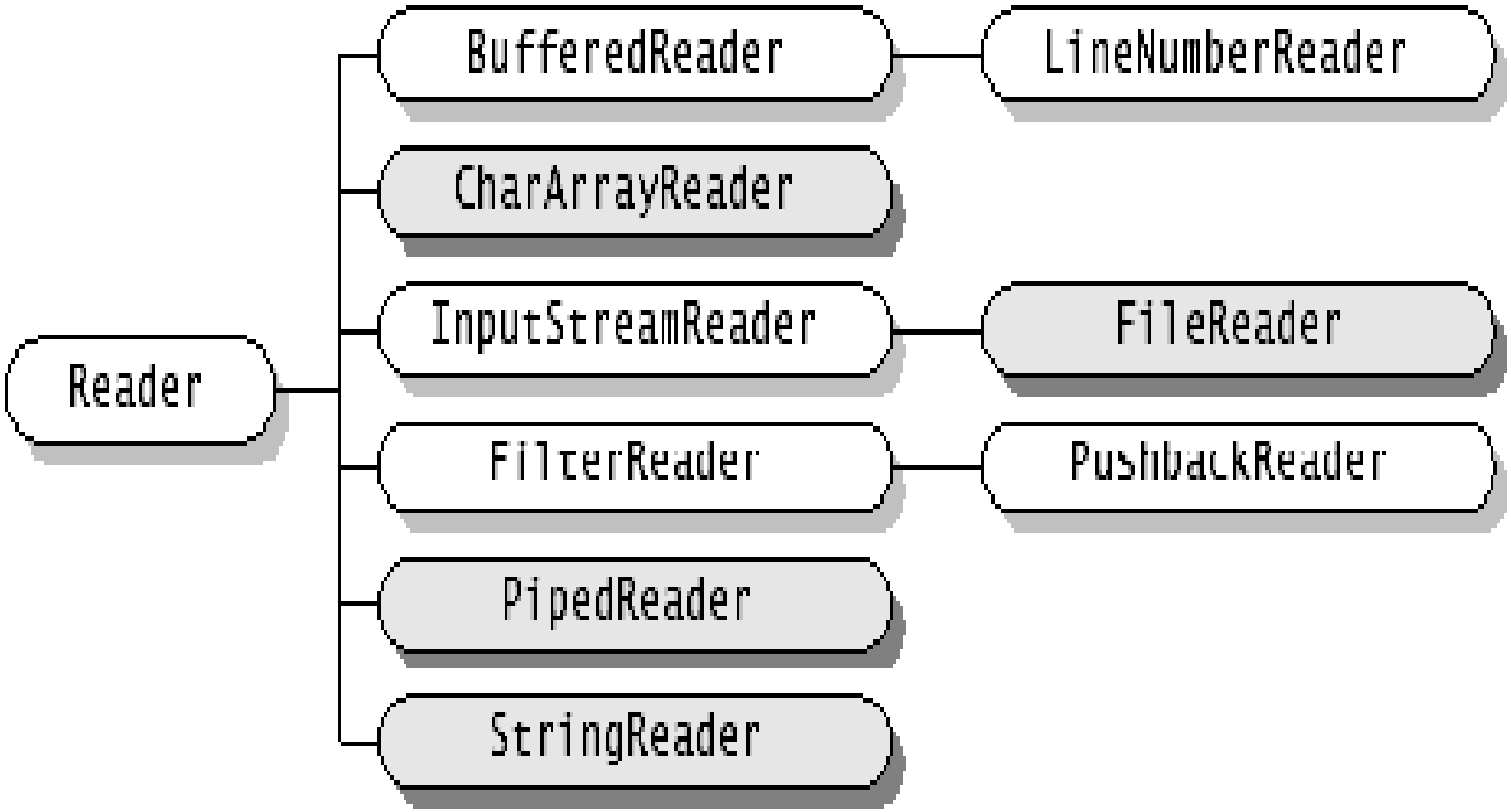
ObjectInputStream

- readObject()

java.io.Serializable interface

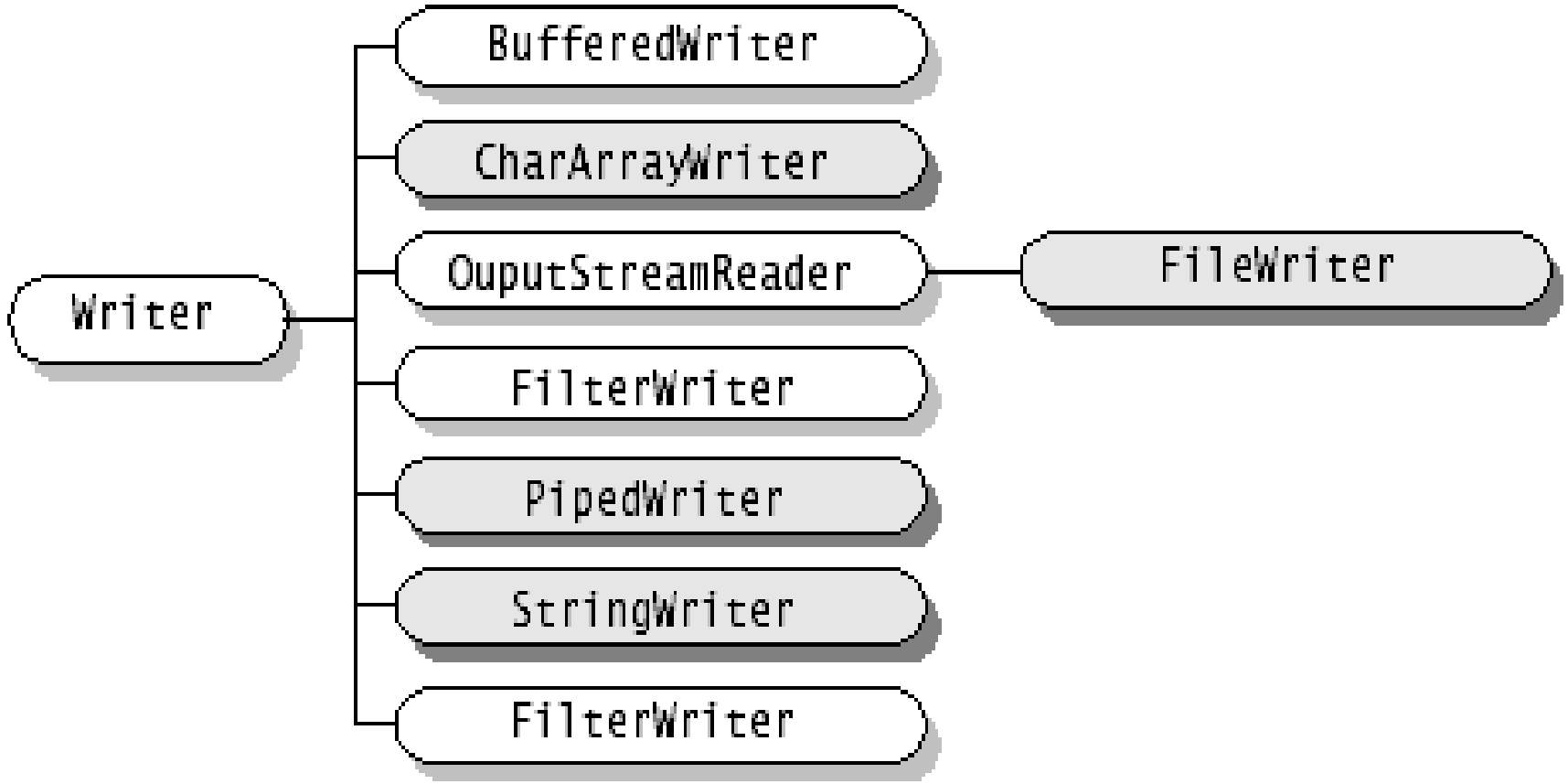
Mastek

Character Stream classes



Mastek

Character Stream classes



Mastek

Reading console input

BufferedReader(Reader inputReader)

InputStreamReader(InputStream inputStream)

int read() throws IOException

String readLine() throws IOException

Mastek

How Annotation Are Used?

Annotations are used to affect the way programs are treated by tools and libraries

Annotations are used by tools to produce derived files

- Tools: Compiler, IDE, Runtime tools
- Derived files : New Java code, deployment descriptor, class files, etc.

Mastek

Why Annotation?

Enables “declarative programming” style

- Less coding since tool will generate the boiler plate code from annotations in the source code
- Easier to change

Eliminates the need for maintaining "side files" that must be kept up to date with changes in source files

- Information is kept in the source file
- Eliminate the need of deployment descriptor

Mastek

How to define Annotation?

Annotation type definitions are similar to normal interface definitions

- An at-sign (@) precedes the interface keyword
- Each method declaration defines an element of the annotation type
- Method declarations must not have any parameters or a throws clause
- Return types are restricted to primitives, String, Class, enums, annotations, and arrays of the preceding types
- Methods can have default values

Mastek

Example

```
/**
* Describes the Request-For-Enhancement(RFE) that
*  led to the presence of the annotated API element.
*/
public @interface RequestForEnhancement {
int id();
String synopsis();
String engineer() default "[unassigned]";
String date(); default "[unimplemented]";
}
```

Mastek

How to use Annotation?

Once an annotation type is defined, you can use it to annotate declarations

- class, method, field declarations

An annotation is a special kind of modifier, and can be used anywhere that other modifiers (such as public, static, or final) can be used

- By convention, annotations precede other modifiers
- Annotations consist of an at-sign (@) followed by an annotation type and a parenthesized list of element-value pairs

Mastek

Usage of Annotation

```
@RequestForEnhancement(
    id = 2868724,
    synopsis = "Enable time-travel",
    engineer = "Mr. Peabody",
    date = "4/1/3007"
    )
    public static void travelThroughTime(Date destination)
    { ... }
```

It is annotating travelThroughTime method

Mastek

Types of Annotation

Three different types of Annotation

- Marker Annotation
- Single Annotation
- Normal Annotation

Mastek

Marker Annotation

An annotation type with no elements

Definition

- /**
- * Indicates that the specification of the annotated API
- * element is preliminary and subject to change.
- */
- public @interface Preliminary { }

Usage – No need to have ()

- @Preliminary public class TimeTravel { ... }

Mastek

Single Value Annotation

An annotation type with a single element

- The element should be named “value”

Definition

- // Associates a copyright notice with the annotated API element.
- public @interface Copyright {String value();}

Usage – can omit the element name and equals sign (=)

- @Copyright("2002 Yoyodyne Propulsion Systems")
- public class OscillationOverthruster { ... }

Mastek

Normal Annotation

We already have seen an example

Definition

- public @interface RequestForEnhancement {
- int id();
- String synopsis();
- String engineer() default "[unassigned]";
- String date(); default "[unimplemented]";
- }

Mastek

Java Database Connectivity

Managing Data

Mastek™

Session Objectives

JDBC Architecture

JDBC Drivers

Using JDBC API’s

Mastek

JDBC Architecture

Mastek

JDBC Drivers

Type IV

Mastek

JDBC API

Packages

- java.sql package
- javax.sql package

Connection

Statement

- Simple
- Prepared
- Callable

ResultSet

Mastek

Basic steps for Connectivity

Load the Database Driver

- Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

Creating a Database Connection

- DriverManager.getConnection("jdbc:odbc:ProductData");

Creating a Statement

- Simple
- Statement st = con.createStatement();
- Prepared
- con.prepareStatement("insert into product values(?,?,?,?)");

Creating a ResultSet

Close the Connection

Mastek

Java 8

Mastek™

What's new in Java 8?

Lambda expression – Adds functional processing capability to Java.

Method references – Referencing functions by their names instead of invoking them directly. Using functions as parameter.

Default method – Interface to have default method implementation.

New tools – New compiler tools and utilities are added like 'jdeps' to figure out dependencies.

Stream API – New stream API to facilitate pipeline processing.

Date Time API – Improved date time API.

Optional – Emphasis on best practices to handle null values properly.

Mastek

Using Lambda Expressions

A lambda expression is characterized by the following syntax.

```
parameter -> expression body
```

Optional type declaration – No need to declare the type of a parameter. The compiler can inference the same from the value of the parameter.

Optional parenthesis around parameter – No need to declare a single parameter in parenthesis. For multiple parameters, parentheses are required.

Optional curly braces – No need to use curly braces in expression body if the body contains a single statement.

Optional return keyword – The compiler automatically returns the value if the body has a single expression to return the value. Curly braces are required to indicate that expression returns a value.

Mastek

Using Interfaces in Functional Programming style

```
interface RequestProcessor{
        processRequest(String name, String data);
}

I1 req1 = new RequestProcessor(String n,String d){
                System.out.println("Data from "+n+":"+d);
                return n;
        };
Req1.processRequest(“Sameer”,”My Message”);
```

Mastek

Using Interfaces in Functional Programming style

```
interface RequestProcessor{
        processRequest(String name, String data);
}
I1 req1 = (n,d) ->
        {
                System.out.println("Data from "+n+":"+d);
                return n;
        };

Req1.processRequest(“Sameer”,”My Message”);
```

Mastek

Functional Interfaces

4 types of Functional Interfaces

a. Supplier<T> { T get()} : provides the values to the caller
b. Consumer<T> { void accept(T t)}: Accepts an object but doesn't provide any return
c. Predicate<T> {boolean test(T t)}: Takes and object, and returns boolean
d. Function<T,R> {R apply(T t)}: Takes an Object and returns applying the operation

Mastek

Default Methods

Java 8 introduces a new concept of default method implementation in interfaces.

```
public interface List {
   default void forEach() {
          // loop over the collection
   }
}
```

An interface can also have static helper methods from Java 8 onwards.

```
   static void printConstants() {
      System.out.println("PI"+PI);
   }
}
```

Mastek

Optional Class

Optional is a container object used to contain not-null objects.

- Assign Optional Object
  - static of(O)
  - static ofNullable(O)
- Check Object is Present
  - isPresent():Boolean
- Get the Object
  - O get()
- If No Object found
  - orElse(O Other)
  - orElseThrow(T Throwable)

Mastek

Collection Streams

Stream represents a sequence of objects from a source, which supports aggregate operations.

Sequence of elements − A stream provides a set of elements of specific type in a sequential manner. A stream gets/computes elements on demand. It never stores the elements.

Source − Stream takes Collections, Arrays, or I/O resources as input source.

Aggregate operations − Stream supports aggregate operations like filter, map, limit, reduce, find, match, and so on.

Pipelining − Most of the stream operations return stream itself so that their result can be pipelined. These operations are called intermediate operations and their function is to take input, process them, and return output to the target. collect() method is a terminal operation which is normally present at the end of the pipelining operation to mark the end of the stream.

Automatic iterations − Stream operations do the iterations internally

Stream Methods

stream() – Returns a sequential stream considering collection as its source.

parallelStream() – Returns a parallel Stream considering collection as its source.

Operations of Stream

- forEach(fx): iterates to each elements passing the object as value
- map(n->{}): calls the operation and returns the value as collection
- filter(condition): performs filter on given criteria
- limit(n): provides the number of elements as given
- skip(n): skips first n elements as given
- sorted(comparator): sorts as per the comparator given
- mapTo<FieldType>(n->{}): reduces to the field given

Mastek

Thank you

Mastek™