Important Links

Thursday, May 11, 2023 11:04 AM

Object Oriented Programming

Thursday, May 11, 2023 11:04 AM

It is
- (may or maynot) ~~real time~~ **Domain** entity
- that has
  - zero or more attributes and
  - zero or more behaviors

Object

~~*instance of*~~

Class

- Blueprint of Object
- No memory unless and instance is created
- An idea that tells you the charactristics of something you interact with.
- not a realtime entity
- doesn't exist
- Multiple Objects can be created.
- Prototype of Object
- Encapsulation of methods and related data.
- ~~A concrete implementation of functionality~~

Encapsulation

HouseBluePrint house1 = new HouseBluePrint( )

```
class HouseBluePrint{
    List<Room> rooms;
    public HouseBluePrint(List<Room> rooms){
        this.rooms=rooms;
    }

}
```

What is a program?

Sequence of instruction to a machine to achive a desirable output

~~A program is a something that takes domain enitity as an input and performs the behaviors associated with that domain and gives desired output.~~

A program is set of objects interacting with each other to achive a goal

What is the goal?
- To represent the domain.

Encapsulation

Thursday, May 11, 2023 12:32 PM

- Combining data and method in single unit
  - Why?
    - To serve data hiding
      - Why?
        - Because we don't want to show every thing to other objects
          - Why?
            - Object should be responsible for it's own state

What is a Class

Thursday, May 11, 2023 2:57 PM

```
class Class{
        String name;
        Field[] fields;
        Method[] methods;
        Constructor[] constructors;
        Class superClass;
        Scope scope;
        Interface[] interfaces;

}
```

Object Oriented Features

Thursday, May 11, 2023 3:05 PM

Abstraction

What?

- separate implementation
- exposing functionality

Why?

How?

- abstract class
- interfaces

Inheritance

What?

- A class inherits the features of other class/interface

Why?

- Reusability of code
- clean code
- modularity???
- versioning
- relational

How?

- extends
- implements

Polymorphism

What?

- Having more than one forms
- Deciding on behavior according to parameter/domain

- Object of different classes can be considered as object of one common class
- Reuse method names.

Why?

- Separation of concerns

How?

- Two types
    - Runtime
        - overriding
    - Compile time
        - overloading

Encapsulation

What?

- binding data to its methods

Why?

- scope limitation.
- data hiding and security
- achieve abstraction by hiding internal details

How?

- defining scopes for attributes and behaviors

Animal object

Thursday, May 11, 2023 3:45 PM

```
Animal tiger=new Animal(AnimalType.Tiger);
Animal snake=new Animal(AnimalType.Snake);

Animal [] animals={tiger,snake};

for(var animal :animals)
    animal.move();
```

```
Tiger tiger=new Tiger();
Snake snake=new Snake();
```

~~Animal [] animals={tiger,snake};~~

~~for(var animal :animals)~~
~~animal.move();~~

Approach #A

- Single class needed
- Can put all animal in a single Animal array

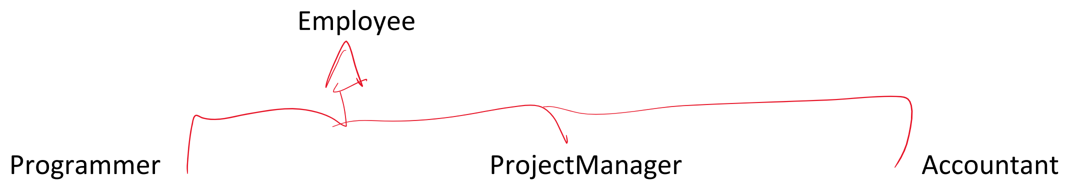
Apporach #B

- Multiple classes needed
- Can't put them in a single array

```
class Animal{

    public String move(){

        switch(animalType){

            case AnimalType.Tiger:
                returns "walks";
            case AnimalType.Snake:
                return "crawls";
            case AnimalType.Eagle:
                return "fly";

        }
    }

}
```

Employee Hierarchy

Thursday, May 11, 2023 4:53 PM

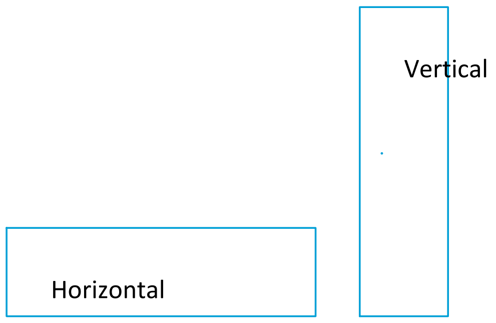


```
Programmer rajiv= new Programmer("Rajiv Bagga");

ProjectManager pm = (ProjectManger) rajiv;
```

Rectangle Square Problem

Thursday, May 11, 2023 5:16 PM

```
class Rectangle{

    double width, height;

    public Rectangle(double width, double height){
        this.width=width;
        this.height=height;
    }

    public double area(){
        return width*height;
    }

    public double perimeter(){
        return 2*(width+height);
    }

    public String draw(){
        return "[" + width+"," + height+ "] ";
    }

    public Oriented getOrientation(){
        if(width>height)
            return Orientation.Horizontal;
        else
            return Oreintation.Vertical;
    }

}
```



```
class Square extends Rectangle{

    public Square(double side){

        super(side,side);
    }

}
```

Parker Pen

Friday, May 12, 2023 10:45 AM

```
class ParkerPen{

    public String useInHand(){
        return "Writing";
    }

    public String useInPocket(){
        return "Status";
    }

}


var p=new ParkerPen();

p.useInHand();

p.useInPocket();
```

```
class ParkerPen{

    public String use(Hand h){
        return "Writing";
    }

    public String use(Pocket p){
        return "Status";
    }

}


var p=new ParkerPen();

Object obj = getHandOrPocket();

//let us assume obj is current Hand

var result = p.use(obj);
```

```
int add(int x,int y){….}

String add(String x, String y){ }
```

```
int add_int_int(int x,int y){….}

String add_String_String(String x, String y){ }
```

```
interface Status{

        String getStatus();
}

interface Pen{
        String write();

}


class ParkerPen implements Pen, Status{

        String getStatus(){…}
        String write(){…}
}

class RenoldsPen implements Pen{

        …
}

class Iphone implements Phone, Status{
```

```
Status  status= getStatusArtifects();

status.getStatus();


Pen pen = getAPen();


pen.write();
```

```
}
```

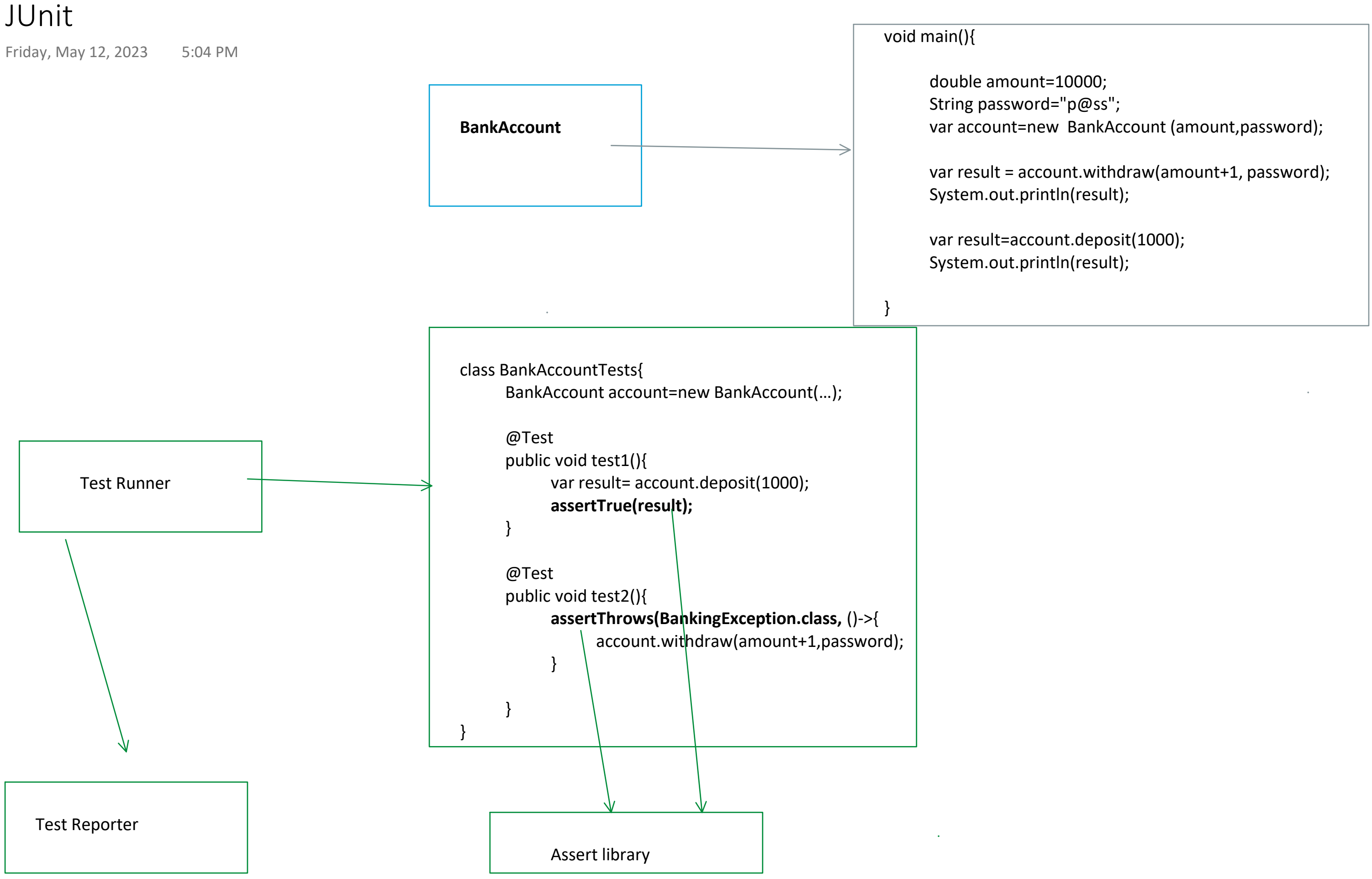
No Interface Model

```
class ParkerPen {

    String getStatus(){…}
    String write(){…}
}

class IPhone {

    String getStatus(){…}

}


class RenoldsPen {

    String getStatus(){…}
    String write(){…}
}
```

```
Object obj = getParkerPenOrRenolds();


var result= obj
                .getClass()
                .getMethod("getStatus")
                .invoke(obj);
```

JUnit

Friday, May 12, 2023 5:04 PM



```
void main(){

        double amount=10000;
        String password="p@ss";
        var account=new  BankAccount (amount,password);

        var result = account.withdraw(amount+1, password);
        System.out.println(result);

        var result=account.deposit(1000);
        System.out.println(result);

}
```

```
class BankAccountTests{
        BankAccount account=new BankAccount(...);

        @Test
        public void test1(){
                var result= account.deposit(1000);
                assertTrue(result);
        }

        @Test
        public void test2(){
                assertThrows(BankingException.class, ()->{
                        account.withdraw(amount+1,password);
                }

        }
}
```

Test Class Practices

- In a typical scenario
- One method may have multiple possible outcomes
    - withdraw
        - fails for
            - insufficient balance
            - invalid password
            - invalid account number
            - negative amount
        - succeed for
            - happy path

1. One Path Per Test

- We don't test all of "withdraw" in a single test case (violates SRP)
- We generallytest one use case in a given test method
-

2. Use Meaningful Phrases as method name. (DAMP)

-
- Since there can be multiple tests for method we can't write
    - withdrawTest
    - or
    - withdrawTest1()
    - withdrawtTest2()

- Descriptiong and Meaningful Phrase
    - withdrawShouldFailForInsufficientBalance()
    - withrawShouldFailForInvalidPassword()
    - withdrawShouldWorkForHappyPath()

3. TEST Phases —> AAA
    - Arrange
        - write the pre-steps before you can test
            - we need to setup BankAccount before we can test for withdraw
    - Act
        - The essential action that we are testing
    - Assert
        - The Assert code to verify you got the expected outcome from the code.

3.1 How to arrange

- We may arrange (initialize the object) at two places

3.1.1. within each test case

```
@Test
public void withdrawShouldReturnBalanceForHappyPath(){
    //arrange
    var account=new BankAccount( amount, password);

    //act
    var result = account.withdraw(amount-1, password);

    //assert
    assertEquals(amount-1, result);
}


@Test
public void depositShouldReturnBalanceForHappyPath(){
    //arrange
    var account=new BankAccount( amount, password);

    //act
    var result = account.withdraw(amount-1, password);

    //assert
    assertEquals(amount-1, result);
}
```

3.1.2 Using a sepcial Initializer method marked with @Before annotation

```
class BankAccountTest{

    @Before
    public void setup(){
        //arrange
        var account=new BankAccount( amount, password);

    }

    @Test
    public void withdrawShouldReturnBalanceForHappyPath(){

        //act
        var result = account.withdraw(amount-1, password);

        //assert
        assertEquals(amount-1, result);
    }


    @Test
    public void depositShouldReturnBalanceForHappyPath(){

        //act
        var result = account.withdraw(amount-1, password);
```

- method marked @Before runs before every test
- In Junit 3- @Before was compulsary
- Junit 4+ we can also initialize the code in constructor instread of @Before

```
        //assert
        assertEquals(amount-1, result);
    }


}
```

4. The fourth "A" of test —> Assume

```
@Test
public void testCase(){
    //Arrange
    //do your setup here

    //Assume
    //make sure your arrangement is right

    //Act
    //now act  on the plan

    //Assert
    //make sure action worked as per plan.

}
```

Best Practice Guidelines

- ### How many asserts per Test method?

Purist Thought

- we should have one Assert per test method
- each test is expected to test just one path (use case) of a method
    - A single test shouldn't test multiple uses cases of a method
- Multiple asserts may mean multiple use case test

Exception to the Rule

- Sometimes a single use case may return / effect multiple states
    - Example
        - A successful bank withdrawal
            - returns status success
            - reduces account balance
- This is not two separate test cases but the outcome of a single path
    - In such cases we can write multple asserts

How do I know it is good to have multiple asserts?

- There should be no gap between asserts.
    - They should follow same "ACT"

Recommendation

- Prefer the rule One assert per test
- Write multiple asserts only when
    - you are sure they are testing same path
    - they are done after a single act

What is TDD or TFD

Friday, May 12, 2023 4:59 PM

- We write ~~tests~~ specs (specificiation) to validate how our future code will work
- We first write specs and then code to adhere to the specs.
- Specs represent the software design that we want to achieve

Life Cycle of a TFD/TDD —> Red, Green, Refactor

Red Phase

- Start with a failing test
- Write a test that fails/errors/compile time error
- Why Fail?
    - This is TDD
    - Actual code doesn't exist yet
        - How can it pass?

- Why TDD?
    - Why write a test when we have nothing to test?
    - This is NOT a test, but a spec.
        - We write what we expect from the final code
    - Red is definining the requirement
        - It is red because it is NOT implemented.
- If we don’t have a failing test, we don't have TDD
    - code already exists and working.

Green Phase

- Write the minimal code to make the test pass
- The minimal code need not be the correct code
    - You may bluff and pass
- How is the minimal code (bluff) useful?
    - It is acknowledging the requirement
    - We received it and we are working on it.
- Why not write the proper code?
    - We may not get the entire specs from a single test method
    - We don't have complete picture yet.
        - It will develop over a period of time

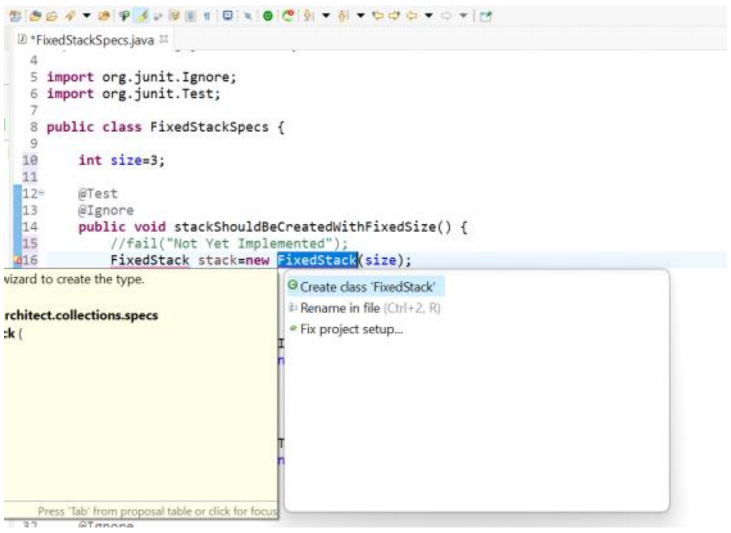
Refactor

- Now replace the dummy code with valid code that can work
    - You don't have to replace it in one goe
    - If specs are correct, the final code may develop automatically
- Every time your make change (refactor)
    - run the code again

- If it passes, you have the right logic
- If it fails you need to fix it

Fixed Stack TDD

Friday, May 12, 2023 5:13 PM



*FixedStackSpecs.java

```java

import org.junit.Ignore;
import org.junit.Test;

public class FixedStackSpecs {

    int size=3;

    @Test
    @Ignore
    public void stackShouldBeCreatedWithFixedSize() {
        //fail("Not Yet Implemented");
        FixedStack stack=new FixedStack(size);
```

izard to create the type.
rchitect.collections.specs
:k {
Create class 'FixedStack'
Rename in file (Ctrl+2, R)
Fix project setup...
Press 'Tab' from proposal table or click for focus

- Here we know the Object's behavior
- We created the Object from a class that doesn't exist
    - Object comes first!
- Now we will ask eclispe to generate the class as requested by the Object

Asserting against Exceptions

Approach #1 Manual Try-catch

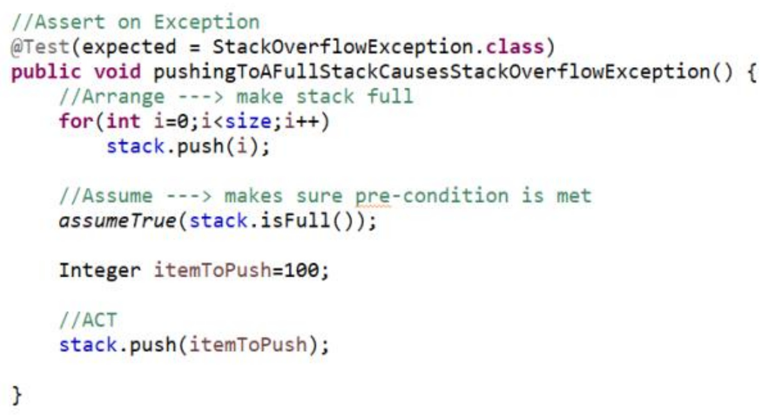
```java
    @Test
    public void pushingToAFullStackCausesStackOverflowException() {
        //Arrange ---> make stack full
        for(int i=0;i<size;i++)
            stack.push(i);

        //Assume ---> makes sure pre-condition is met
        assumeTrue(stack.isFull());

        Integer itemToPush=100;

        //ACT
        try {
            stack.push(itemToPush);
            //Assert
            fail("expected exception 'StackOverflow was not thrown'");
        } catch(StackOverflowException ex) {

            //Test passed. do nothing.
            assertEquals(itemToPush, ex.getItemPushed());
        }
    }
```

- since we except push to throw
- If we it doesn't throw and we continue inside try
    - we fail the test explicitly
- if exception was thrown we reach catch block
    - since exception was expcected test passed
    - Do Nothing or
        - Assert against exception values

Option #2 —> If all you need to test is exception is thrown

- and you don't want to assert against exception object itself
- Works only in Junit 4
- Here we defined expcted exception inside @Test annotation

- ○ code is simple to read
- ○ we can't test against exception object

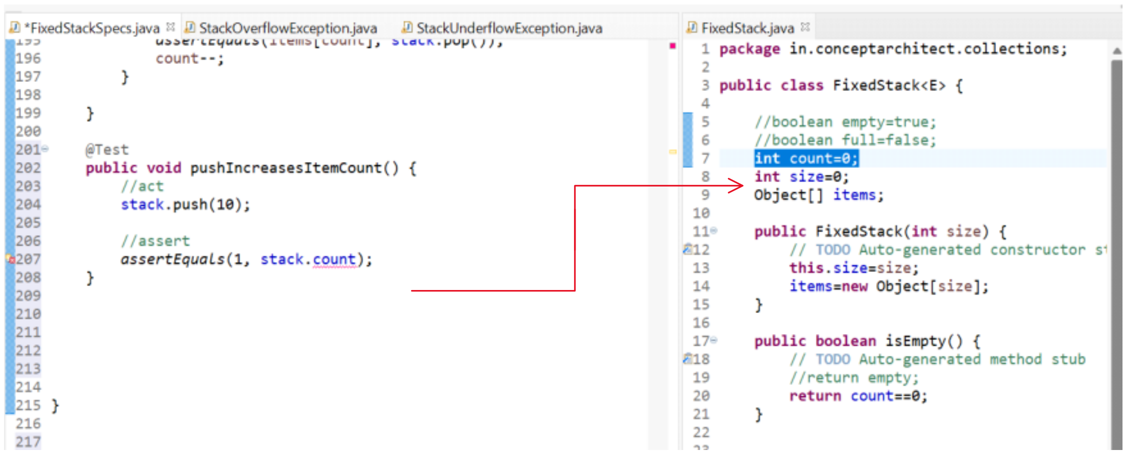


```java
//Assert on Exception
@Test(expected = StackOverflowException.class)
public void pushingToAFullStackCausesStackOverflowException() {
    //Arrange ---> make stack full
    for(int i=0;i<size;i++)
        stack.push(i);

    //Assume ---> makes sure pre-condition is met
    assumeTrue(stack.isFull());

    Integer itemToPush=100;

    //ACT
    stack.push(itemToPush);

}
```

Testing internal Implementation

Saturday, May 13, 2023 12:05 PM

- sometimes we need to test against internal implementation of a code and NOT only public methods



*FixedStackSpecs.java | StackOverflowException.java | StackUnderflowException.java

```java
                assertEquals(items[count], stack.pop());
                count--;
            }

    }

    @Test
    public void pushIncreasesItemCount() {
        //act
        stack.push(10);

        //assert
        assertEquals(1, stack.count);
    }







}
```

FixedStack.java

```java
package in.conceptarchitect.collections;

public class FixedStack<E> {

    //boolean empty=true;
    //boolean full=false;
    int count=0;
    int size=0;
    Object[] items;

    public FixedStack(int size) {
        // TODO Auto-generated constructor st
        this.size=size;
        items=new Object[size];
    }

    public boolean isEmpty() {
        // TODO Auto-generated method stub
        //return empty;
        return count==0;
    }
```

- Since internal implementation is not accessible by other classes It can't be easily accessed.

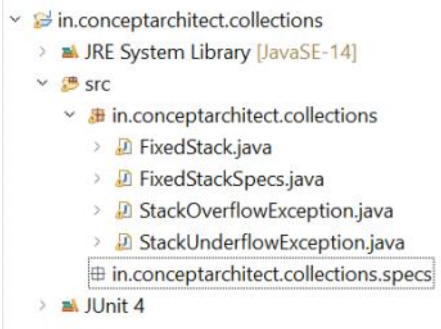
- We can do it in two different ways

1. Testing Package scoped members

- Step #1
  - It is often a good approach to prefer "package scope" over "private scope"
    - They can be accessed by your own code within the package
    - They are still protected from outsider code
- Step #2
  - Write your test in the same package and not in some other package

Problem

- Now both my tests/specs and actual code is the same folder
- It becomes difficult to organize large number of classes and their tests/specs in the same folder



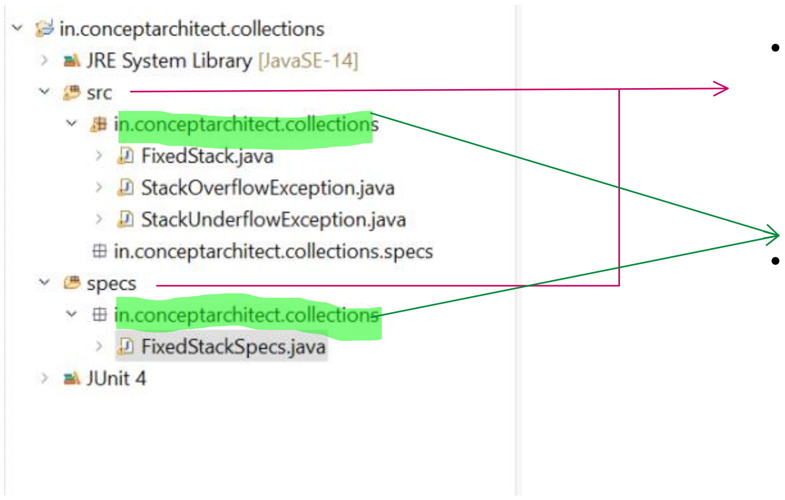
```
in.conceptarchitect.collections
  JRE System Library [JavaSE-14]
  src
    in.conceptarchitect.collections
      FixedStack.java
      FixedStackSpecs.java
      StackOverflowException.java
      StackUnderflowException.java
    in.conceptarchitect.collections.specs
  JUnit 4
```

Solution

1. Create Two separate source Folder (that are added to the class path)
   - src
     - to hold actual source code
   - specs
     - to hold the specs file

2. Now add same package within both the source folder (class path)

3. Add specs under specs folder



```
in.conceptarchitect.collections
  JRE System Library [JavaSE-14]
  src
    in.conceptarchitect.collections
      FixedStack.java
      StackOverflowException.java
      StackUnderflowException.java
    in.conceptarchitect.collections.specs
  specs
    in.conceptarchitect.collections
      FixedStackSpecs.java
  JUnit 4
```

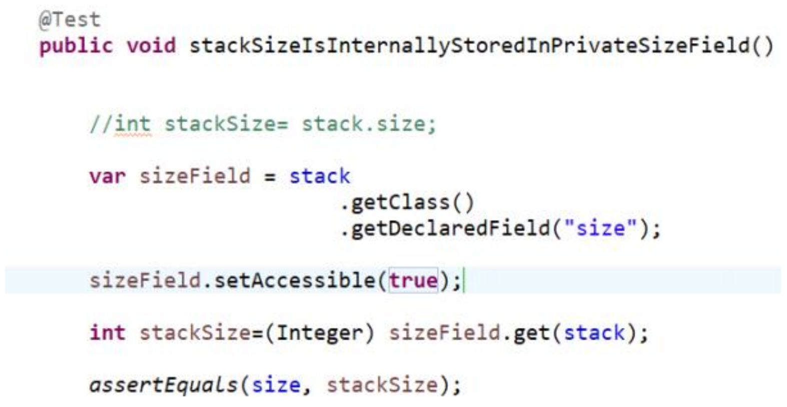
- different physical folders organizes the buisness class and specs/tests classes separately

- Same package in both folder allows package scopes to work

Testing against private fields

- Very rare
- Prefer making those implementation package
- Private can't be accessed by any other classes even within the same package directly

- ### But if you must
  - You can use reflection to test the private members

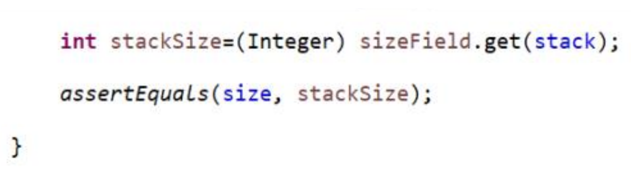


```
@Test
public void stackSizeIsInternallyStoredInPrivateSizeField()

    //int stackSize= stack.size;

    var sizeField = stack
                        .getClass()
                        .getDeclaredField("size");

    sizeField.setAccessible(true);

    int stackSize=(Integer) sizeField.get(stack);

    assertEquals(size, stackSize);
```

- get the private field

- make is accessible

- get the value of the field



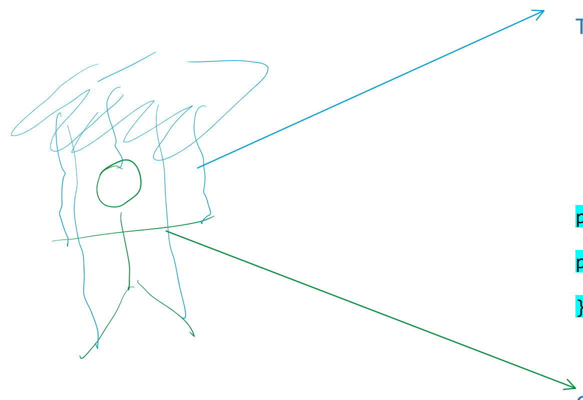
```
    int stackSize=(Integer) sizeField.get(stack);

    assertEquals(size, stackSize);

}
```

- make is accessible

- get the value of the field
- assert against the field

Multi-threading

Saturday, May 13, 2023 12:39 PM



Thread

- A thread is a controller for a part (sub task)
- It makes the sub task execute independently on a separate path
- This is typically a predefined language feature

```
package java.lang;

public class Thread{

}
```

Controlled Item (Task)

- Represents what (task) thread controls.
- This is typically a user defined business logic

```
package java.lang;

public interface Runnable{

    void run();
}
```

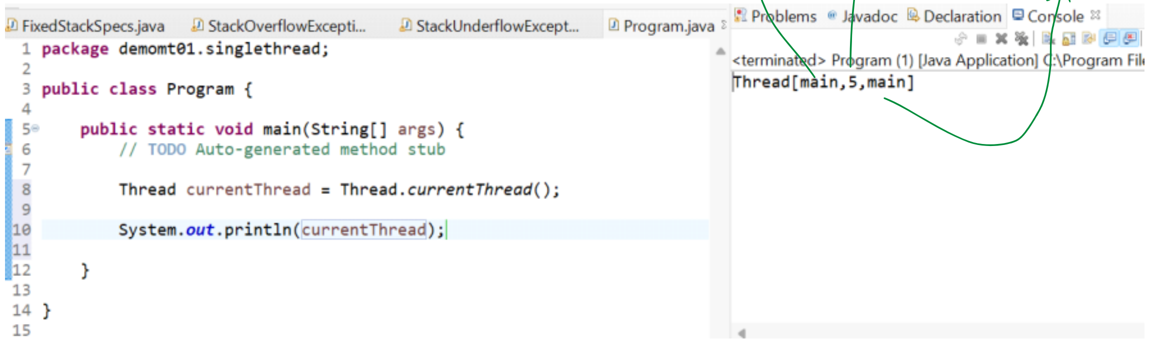
An application is either Multi-threaded or Single Threaded

- There is no unthreaed application
- All Java Application has at least one thread
  - known as main thread

Thread Name

Thread Group Name

Thread Priority
- varies between
  - 1—> least
  - 5 —> normal
  - 10 —>highest



```
package demomt01.singlethread;

public class Program {

    public static void main(String[] args) {
        // TODO Auto-generated method stub

        Thread currentThread = Thread.currentThread();

        System.out.println(currentThread);

    }

}
```

<terminated> Program (1) [Java Application] C:\Program Fil

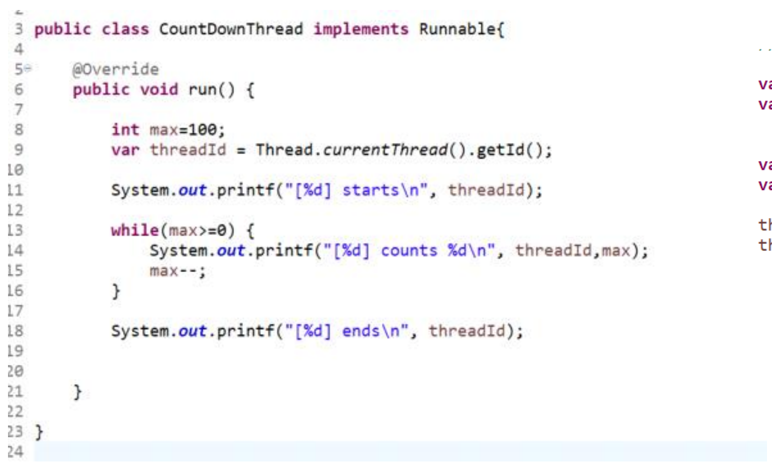
```
Thread[main,5,main]
```

Creating User Defined Thread

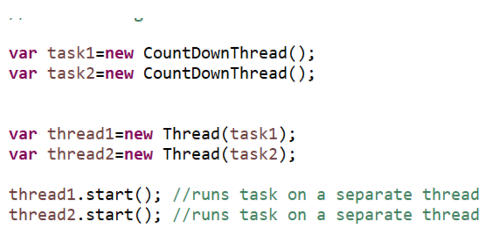
Option #1 Implement Runnable

1. Create your task by implementing Runnable
2. Create Thread Object and pass your task as paramter
3. Start the Thread

```
public class CountDownThread implements Runnable{

    @Override
```



```java
public class CountDownThread implements Runnable{

    @Override
    public void run() {

        int max=100;
        var threadId = Thread.currentThread().getId();

        System.out.printf("[%d] starts\n", threadId);

        while(max>=0) {
            System.out.printf("[%d] counts %d\n", threadId,max);
            max--;
        }

        System.out.printf("[%d] ends\n", threadId);


    }

}
```

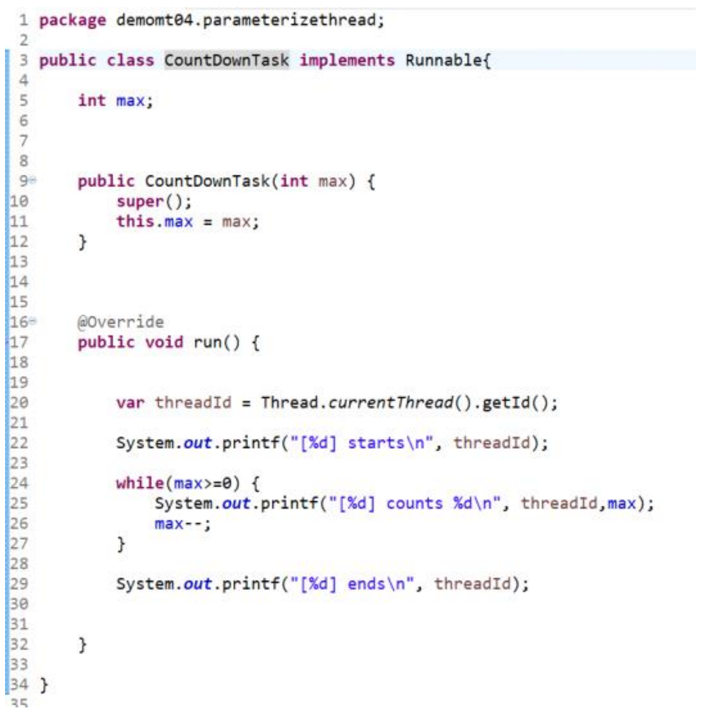


```java
var task1=new CountDownThread();
var task2=new CountDownThread();


var thread1=new Thread(task1);
var thread2=new Thread(task2);

thread1.start(); //runs task on a separate thread
thread2.start(); //runs task on a separate thread
```

How to pass parameters to Threaded Task

- Runnable interface run() doesn't take any parameter
- How do we pass a parameter.

Solution —> we can pass the parameter to a the constructror of the class and store in field



```java
package demomt04.parameterizethread;

public class CountDownTask implements Runnable{

    int max;



    public CountDownTask(int max) {
        super();
        this.max = max;
    }



    @Override
    public void run() {


        var threadId = Thread.currentThread().getId();

        System.out.printf("[%d] starts\n", threadId);

        while(max>=0) {
            System.out.printf("[%d] counts %d\n", threadId,max);
            max--;
        }

        System.out.printf("[%d] ends\n", threadId);


    }

}
```

Creating User Defined Thread Option #2 —> Extend Thread

- Thread class itself implments Runnable
- We can write our logic in a class after extending Thread

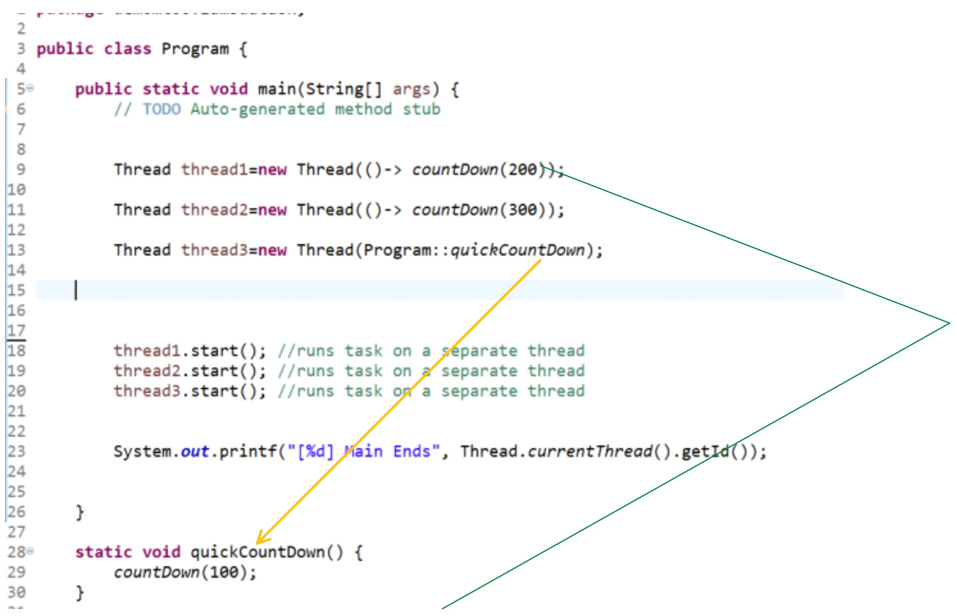
```
interface Runnable{

    void run();
}


class Thread  implements Runnable{
    Runnable runnable;

    public Thread(Runnable runnable){
        this.runnable=runnable;
    }


    public Thread(){
        this.runnable=this;
    }

    public void run(){ }

    public void start(){
        //create a new Thread at OS level

        runnable.run();

    }
class MyTask implements Runnable{
}
    public void run(){
        //user defined logic
    }
}

void main(){

    MyTask task = new MyTask();

    Thread t= new Thread(task);

    t.start();

}
```

```
class MyThread extends Thread{

    public void run(){
        //user defined logic
    }

}



void main(){

    MyThread t = new MyThread();

    ~~Thread t= new Thread(task);~~

    t.start();

}
```

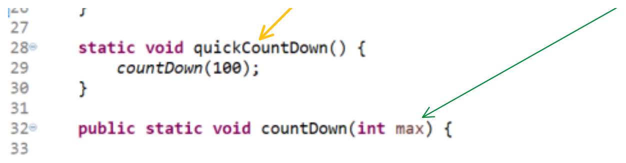
Which one to use?

- Thread implements Runnable violates SRP
    - Thread is a controller
    - Runnable is a task
- We should avoid extending Thread to describe user defined business

Implement Runnable using lambda/method references



```

public class Program {

    public static void main(String[] args) {
        // TODO Auto-generated method stub


        Thread thread1=new Thread(()-> countDown(200));

        Thread thread2=new Thread(()-> countDown(300));

        Thread thread3=new Thread(Program::quickCountDown);




        thread1.start(); //runs task on a separate thread
        thread2.start(); //runs task on a separate thread
        thread3.start(); //runs task on a separate thread


        System.out.printf("[%d] Main Ends", Thread.currentThread().getId());


    }

    static void quickCountDown() {
        countDown(100);
    }
```



```java
    }

    static void quickCountDown() {
        countDown(100);
    }

    public static void countDown(int max) {
```

Thread Methods

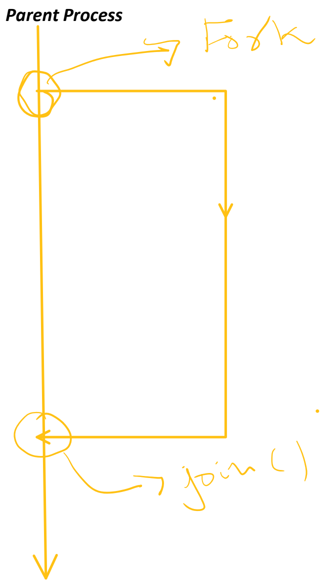
- isAlive
    - returns boolean if given thread is running
- Sleep
    - sleeps for a given number of millisecond
- join

- Makes current thread sleep till the other thread (on which you called join) finished.

```java
thread1.join(); //current thread sleeps till thread1 finishes
```

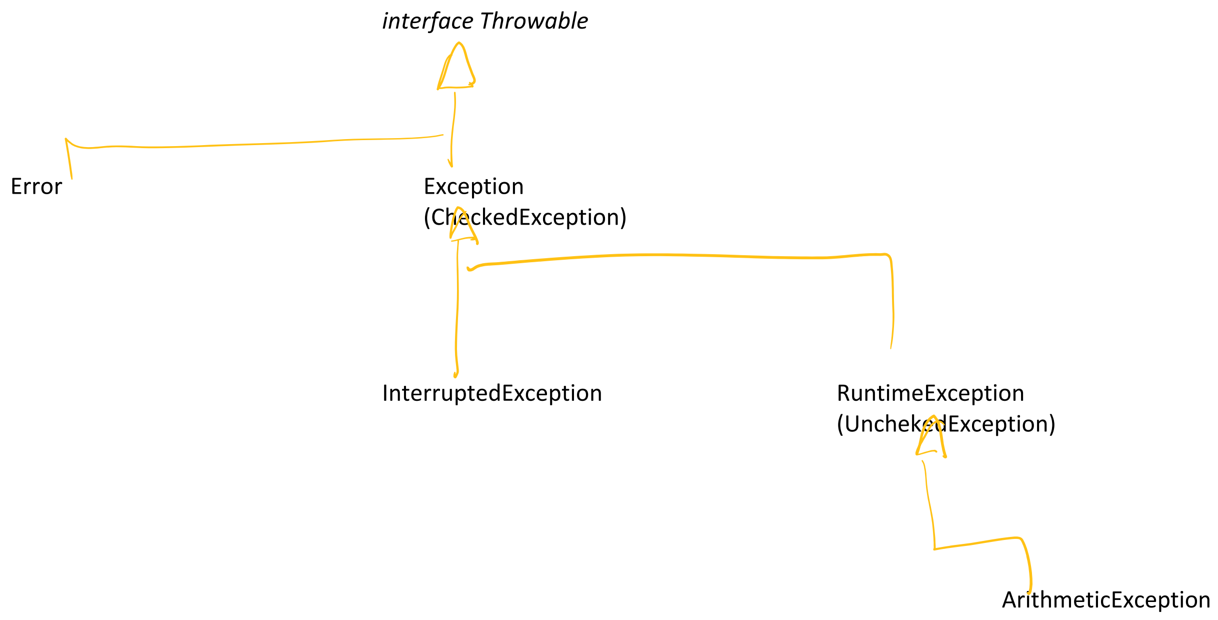
- Because it is a form of sleep, it can be Interrupted.
    - you need either try-catch or throws
    - or wrap in unchecked exception

Unix Multitasking Architecture



Exception

Saturday, May 13, 2023 2:57 PM



- ATM.mainMenu()
    - ATM.userMenu()
        - ATM.withdrawMenu()
            - BankService.withdraw()
                - BankAccount.withdraw()
                    - BankAccount.authenticate() ————> **throws new InvalidCredentialException()**

Thread Memory Model

Saturday, May 13, 2023 4:08 PM

- typically an application's memory is divided in 4 parts.
  - Code Segment
    - This is where our code is loaded
      - classes
      - methods
  - Data Segment
    - contains fixed memory elements like
      - static
      - final
  - Stack Segment
    - contains method locals like
      - method parameter
      - local variables declared within the method
      - return value
  - Heap Segment
    - contains dynamic allocated memory allocted using new

Multi-threading

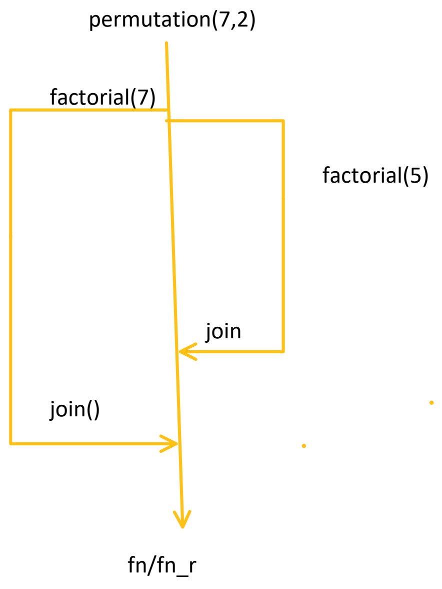
- Each thread maintains a separate "Stack Segment"
  - typically 2MB
- Each thread shares the remaining segment

Thread Return

Saturday, May 13, 2023 4:12 PM

How can we return a value from a Thread once it finishes.

```
int permutation(int n, int r){

     int fn= factorial(n);

     int fn_r=factorial(n-r);


     return fn/fn_r;
}
```



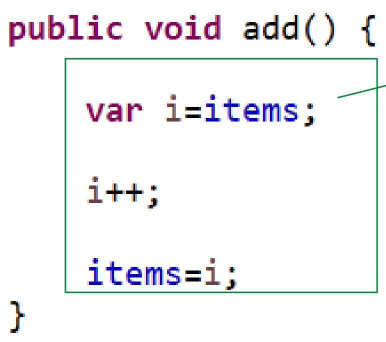
Shared Resources

```
public void add() {

     var i=items;

     i++;

     items=i;
}
```

- multiple threads reaches here
    - assume value items at this point in 10
    - each gets local i=10
- each increment i=11
- each update shared items =11
- After 4 additional the value should have been 14
    - it ends up being 11

Locking The shared resources

- We need to make sure that shared resources are accessed one at a time.
- Different langauge or framework use different term to represent same idea



```
public void add() {
    var i=items;

    i++;

    items=i;
}
```

- We have a
  - Critical Section of code
  - That must be accessed in
    - Mutually Exclusive (Mutex) manner
  - We do it by Monitoring or Locking resource
  - to get **syncrhonized** access

90360 VIVEK

S