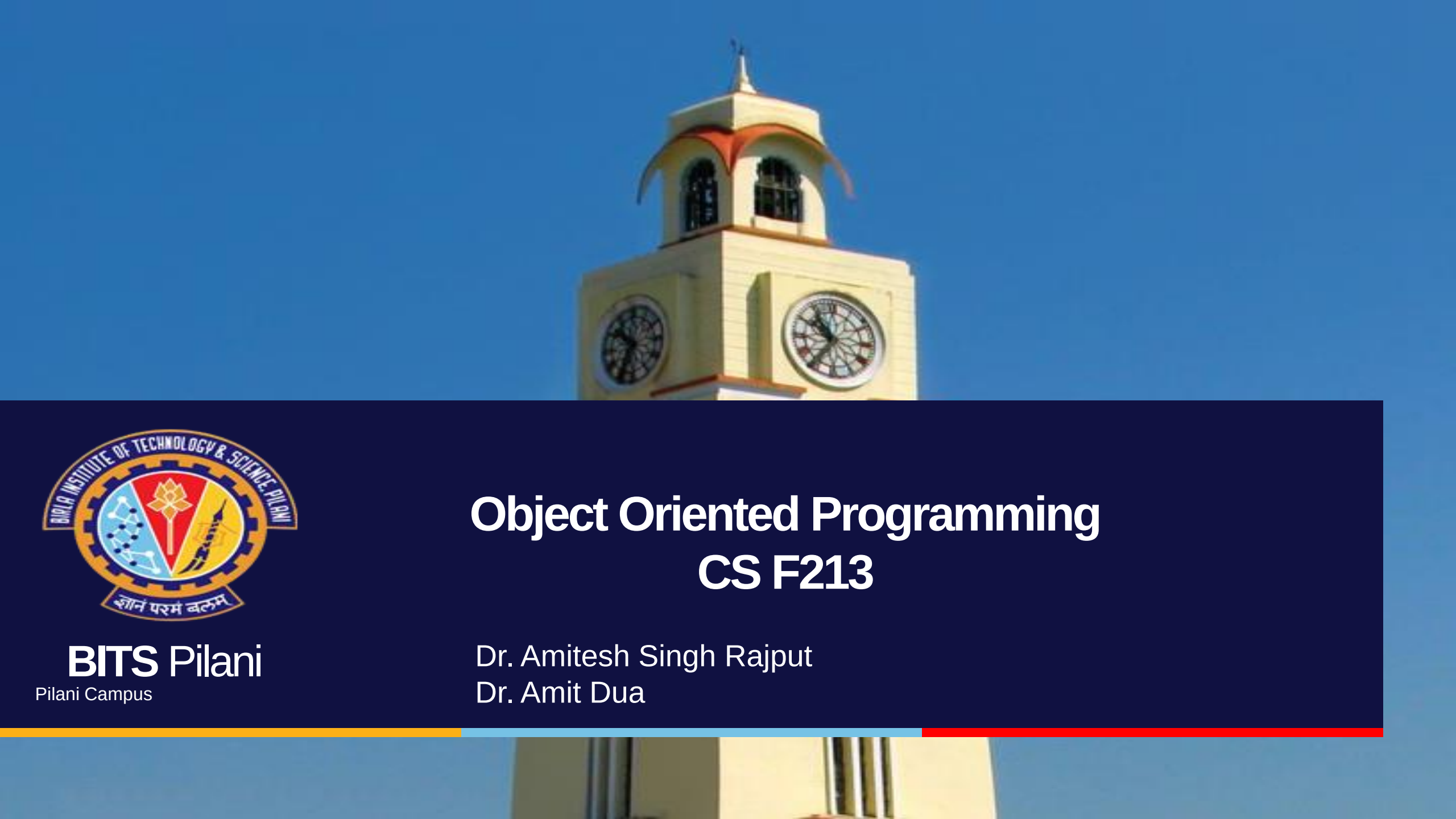


Object Oriented Programming
CS F213

BITS Pilani
Pilani Campus

Dr. Amitesh Singh Rajput
Dr. Amit Dua

innovate achieve lead

BITS Pilani

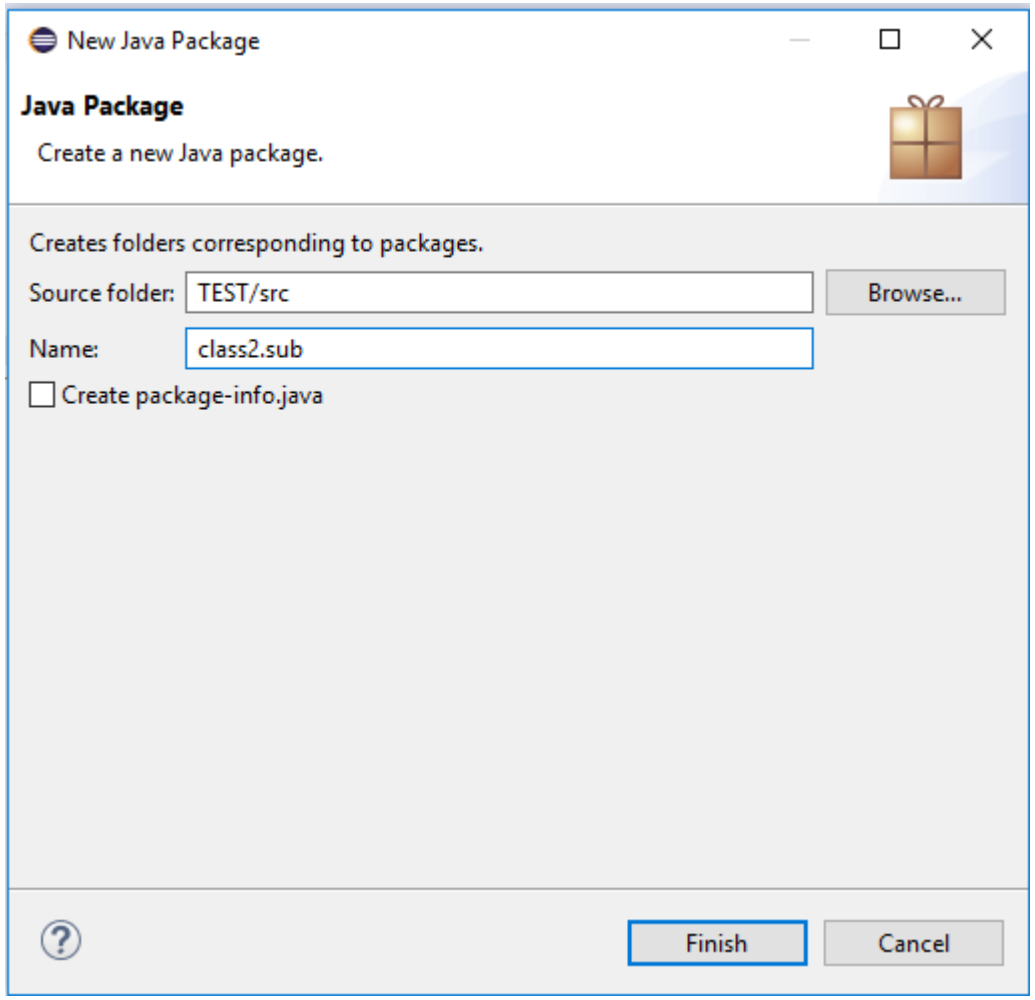
Pilani Campus

Packages

Create a package & sub package

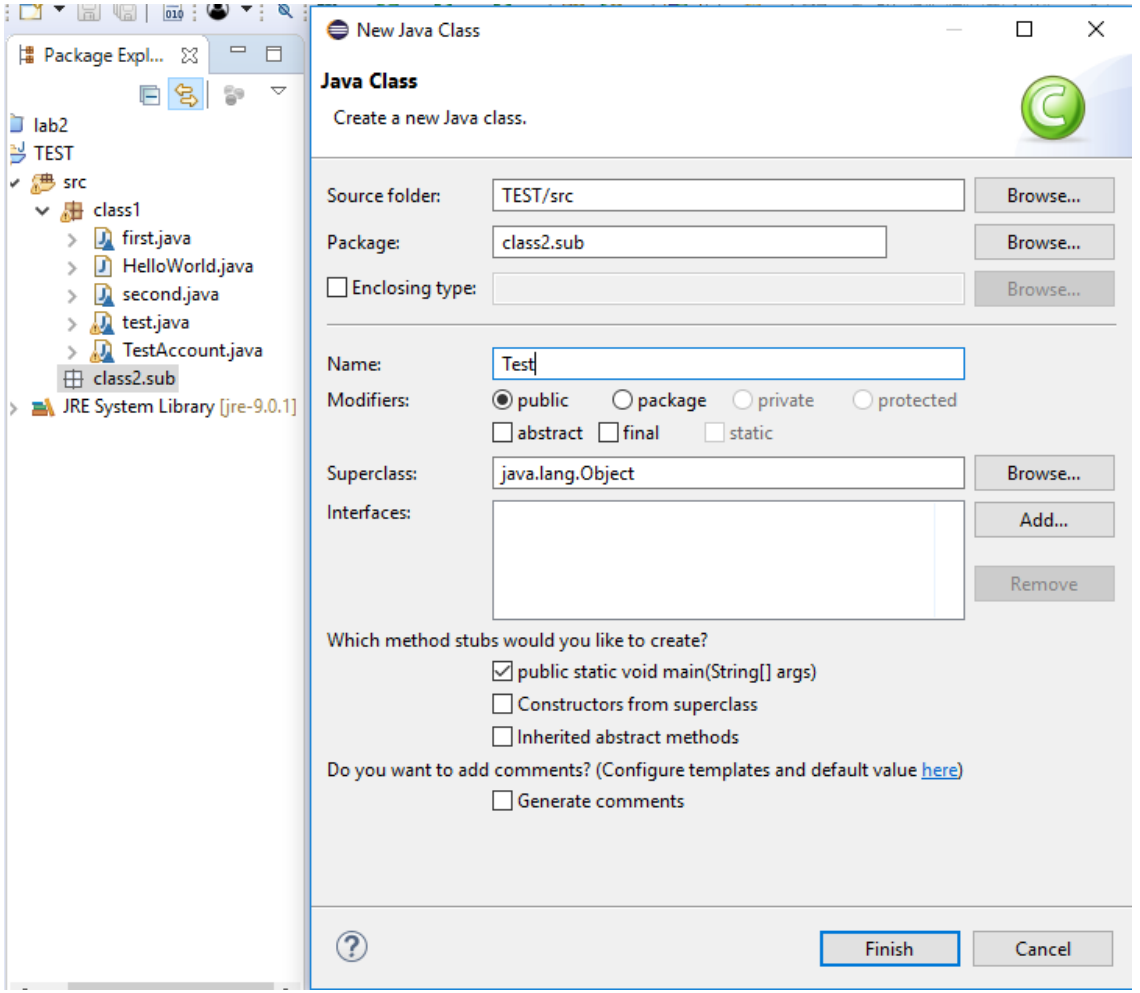
innovate achieve lead

`Project → New → Package`



Create a class within the package

`Package → New → class`



Importing a package

```
package class1;

public class HelloWorld{
 public void show(){
  System.out.println("Within class 1");
  }
}
```

```
package class2.sub;
import class1.*;

public class Test {
 public static void main(String[] args) {
  HelloWorld h = new HelloWorld();
  h.show();
 }
}
```

Importing a class

```
package class1;

public class HelloWorld
{
 public void show() {
  System.out.println("Within class
   1's show");
  }
}
```

```
package class2.sub;
import class1.HelloWorld;

public class Test {
 public static void main(String[] args) {
 HelloWorld h = new HelloWorld();
 h.show();
}
}
```

Take Home Exercise: Learn how to execute the same code from the command prompt.

innovate achieve lead

BITS Pilani

Pilani Campus

Generics

Generics

- Generics mean **parameterized types**. Similar to templates in C++.
- Allows type (Integer, String, etc.) to be a parameter to methods, classes and interfaces.
- <> is used to specify the parameter types.
- To create objects use the following syntax:

```
BaseType <Type> obj = new BaseType <Type>()
```

Note: In Parameter type we can not use primitives like 'int', 'char' or 'double'.

Generic Class - Example


```
class Identity<T>{
    T obj;
    Identity(T obj){
        this.obj = obj;
    }
    public T getObject(){
        return this.obj;
    }
}

class Test {
    public static void main (String[] args){
      Identity <Long> number = new Identity<Long>(9999955555L);
      System.out.println(number.getObject());

      Identity <String> name = new Identity<String>("Ankit");
      System.out.println(name.getObject());
    }
}
```

Output

9999955555

Ankit

- Generics in Java was added to provide **type-checking at compile time and it has no use at run time**.
- Java compiler uses **type erasure** feature to remove all the generics type checking code in byte code and insert type-casting if necessary.
- Type erasure ensures that no extra classes are created.
- Generics incur no runtime overhead.

Multiple Type Parameters


```
class Identity<T,U> {
    T obj1; U obj2;
    Identity(T obj1, U obj2){
        this.obj1 = obj1;
        this.obj2 = obj2;
    }
    public void printObject(){
        System.out.println(this.obj1);
        System.out.println(this.obj2);
    }
}
class Test {
    public static void main (String[] args){
        Identity <String, Integer> I1 = new Identity<String, Integer>("Ankit", 20171007);
        Identity <Integer, String> I2 = new Identity<Integer, String>(20171007, "Ankit");
        I1.printObject();
        I2.printObject();
    }}
```

Output

Ankit
20171007
20171007
Ankit

Generic Functions

```
class Identity {
    public <T> void printObject(T obj){
        System.out.println(obj.getClass().getName()+ " "+obj);
    }
}

class Test {
    public static void main (String[] args){
        Identity I1, I2;
        I1 = new Identity();
        I2 = new Identity();
        I1.printObject(20071007);
        I2.printObject("Ankit");
    }
}
```

Output

java.lang.Integer 20071007
java.lang.String Ankit

Generic Functions with generic return type


```
class Identity{
    public <T> T printObject(T obj){
        return obj;
    }
}

class Test {
    public static void main (String[] args){
        Identity I1, I2;
        I1 = new Identity();
        I2 = new Identity();
        System.out.println(I1.printObject(20071007));
        System.out.println(I2.printObject("Ankit"));
    }
}
```

Generics in Interfaces


```
interface DemoInterface <T1, T2>{       //Generic interface definition
    T2 doSomeOperation(T1 t);
    T1 doReverseOperation(T2 t);
}

class DemoClass implements DemoInterface <String, Integer>{
    public Integer doSomeOperation(String t){
        …
    }
    public String doReverseOperation(Integer t){
        …
    }
}
```

Bound Type with Generics

- Used to **restrict the types that can be used as arguments** in a parameterized type.
  - E.g: Method operating on numbers should accept the instances of the Number class or its subclasses.
- Declare a bounded type parameter
  - List the type parameter's name.
  - Along by the extends keyword.
  - And by its upper bound.

Bound Type - Example

```
class Identity <T extends Number>{
   T obj;
   Identity(T obj){
        this.obj = obj;
        System.out.println("Double value" + obj.doubleValue());
   }
   public T getObject(){
        return this.obj;
   }
}
```

```
class Test {
  public static void main (String[] args){
    Identity <Integer> iObj = new Identity<Integer>(207);
    System.out.println(iObj.getObject());
    Identity <Float> fObj = new Identity<Float>(20.7f);
    System.out.println(fObj.getObject());
    Identity <String> sObj = new Identity<String>("Ankit");
    System.out.println(sObj.getObject());
  }}
```

Generics and Inheritance


```
class MyClass<T>{ }

class Main {
    public static void main(String[] args) {
        String str = "abc";
        Object obj = new Object();
        obj = str;
       // works because String is-a Object (inheritance)
        }
}
```

Generics and Inheritance


```
class MyClass<T>{ }

class Main {
    public static void main(String[] args) {
        MyClass<String> myClass1 = new MyClass<String>();
        MyClass<Object> myClass2 = new MyClass<Object>();
        myClass2 = myClass1;
        // compilation error since MyClass<String> is not a MyClass<Object>
        }
}
```

What are not allowed with Generics?


```
public class GenericsExample<T>
{
   private static T member; //This is not allowed
}
```

```
public class GenericsExample<T>
{
   public GenericsExample(){
      new T();
   }
}
```

What are not allowed with Generics?


```
final List<int> ids = new ArrayList<>();     //Not allowed

final List<Integer> ids = new ArrayList<>(); //Allowed
```

```
// causes compiler error
public class GenericException<T> extends Exception {}
```

Bounded Types – Additional Info

```
class A { }

class B { }

interface C { }

interface D { }

class E<T extends A & C & D> {

}
```

- **T** is bounded by a class **A** and interface **C** and **D**.
- Type argument passed to **T** must be a subclass of **A** and have implemented **C** and **D**

innovate achieve lead

BITS Pilani

Pilani Campus

Thank You!