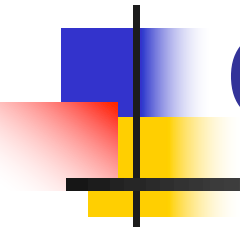
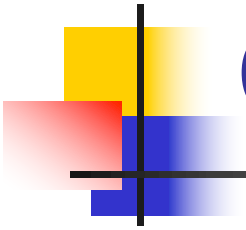
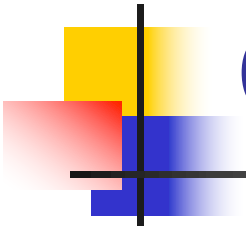


Generics



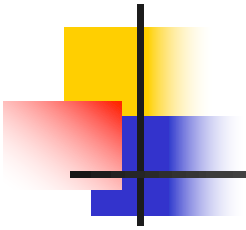
Generics

- Added to Java v5.0
- Generics = class and method definitions may include parameters for types
- Generic class – type parameter allows one to write code that applies to any class
- Ex. list of items of type T
  - When T=Double, it's a list of Doubles, etc.



Generics

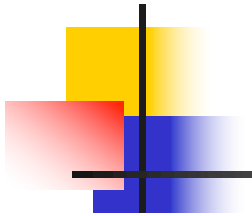
- Classes and methods can have a type parameter (actually, more than one).
- The type parameter can be any reference (class) type.
- (started with Java version 5.0; in C++ as well but much restricted in Java)
- Generic class = parameterized class = class with a parameter for a type



Generics

- A class definition with a type parameter is stored in a file and compiled just like any other class.
- Once a parameterized class is compiled, it can be used like any other class.
  - However, the class type plugged in for the type parameter must be specified before it can be used in a program.
  - Doing this is said to *instantiate* the generic class.

```
Sample<String> object = new Sample<String>();
```

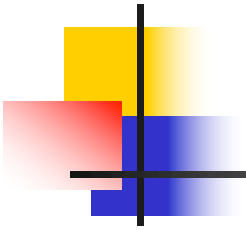


Generics

Display 14.4 A Class Definition with a Type Parameter

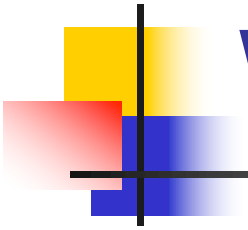
```
1   public class Sample<T>
2   {
3       private T data;

4       public void setData(T newData)
5       {
6           data = newData;
7       }

8       public T getData()
9       {
10          return data;
11      }
12  }
```

T is a parameter for a type.



Generics

- A class that is defined with a parameter for a type is called a generic class or a parameterized class
  - The type parameter is included in angular brackets after the class name in the class definition heading.
  - Any non-keyword identifier can be used for the type parameter, but by convention, the parameter starts with an uppercase letter.
  - The type parameter can be used like other types used in the definition of a class.



What is Generics

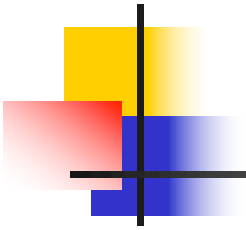
- Collections can store Objects of any Type
- Generics restricts the Objects to be put in a collection
- Generics ease identification of runtime errors at compile time

How is Generics useful?

Consider this code snippet

```
List v = new ArrayList();
v.add(new String("test"));
Integer i = (Integer) v.get(0);
  // Runtime error . Cannot cast from String to Integer
```

This error comes up only when we are executing the program and not during compile time.



How does Generics help

The previous snippet with Generics is

```
List<String> v = new ArrayList<String>();
v.add(new String("test"));
Integer i = v.get(0);
// Compile time error. Converting String to Integer
```

- The compile time error occurs as we are trying to put a String and convert it to Integer on retrieval.
- Observe we don’t have to do an explicit cast when we invoke the get method.
- We can also use interfaces in Generics

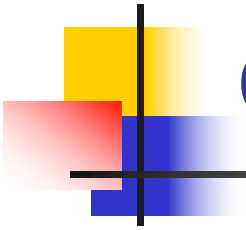
Generic methods

```
public class Utility {
    …
    public static <T> T getMidpoint ( T[] a ) {
        return a[ a.length/2 ];
    }
    public static <T> T getFirst ( T[] a ) {
        return a[0];
    }
    …
}
```

Generic methods

```
public class Utility {
    public static <T> T getMidpoint ( T[] a ) {
        return a[ a.length/2 ];
    }
    public static <T> T getFirst ( T[] a ) {
        return a[0];
    }
}
//usage:
String  midString = Utility.<String>getMidPoint( b );
double  firstNumber = Utility.<Double>getFirst( c );
```

Generic (parameterized) classes



```
public class Sample<T> {
   private T  data;
   public void setData ( T newData ) {
        data = newData;
   }
   public T getData ( ) {
        return data;
   }
}
```

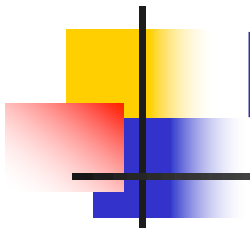
Generic (parameterized) classes

```
public class Sample<T> {
    private T  data;
    public void setData ( T newData ) {
        data = newData;
    }
    public T getData ( ) {
        return data;
    }
}
//usage:
Sample<String>  sample= new Sample<String>();
sample.setData( “Hello” );
```

Generic class for ordered pairs

```
Pair<String>  secretPair
        = new Pair<String>( “Happy”, “Day” );
Pair<Integer>  pairOfDice
        = new Pair<Integer>( new Integer(2),
                             new Integer(3) );
Pet  male = new Pet();
Pet  female = new Pet();
Pair<Pet>  breedingPair
        = new Pair<Pet>( male, female );
```

Defining the ordered pair class



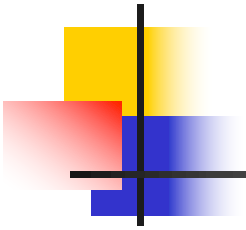
```
public class Pair<T> {
    private T  first;
    private T  second;
    public Pair ( ) {
            first = null;
            second = null;
    }
    public Pair ( T f, T s ) {
            first = f;
            second = s;
    }
    …
}
```

Defining the ordered pair class

```
public class Pair<T> {
    private T  first;
    private T  second;
    …
    public boolean equals ( Object other ) {
        if (other==null)                          return false;
        if (getClass() != other.getClass())       return false;
        Pair<T>  o = (Pair<T>)other;
        return first.equals(o.first) && second.equals(o.second);
    }
    …
}
```

More then one type parameter can be specified

```
Pair<String,Integer>
   p = new Pair<String,Integer>( “Kyle Jones”,
                                 new Integer(123456789) );
```



Defining the ordered pair class

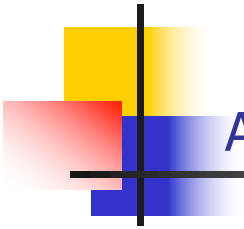
```
public class Pair<T1,T2> {
    private T1  first;
    private T2  second;
    public Pair ( ) {
        first = null;
        second = null;
    }
    public Pair ( T1 f, T2 s ) {
        first = f;
        second = s;
    }
    …
}
```

Defining the ordered pair class

```
public class Pair<T1,T2> {
    private T1   first;
    private T2   second;
    …
    public T1 getFirst ( ) {  return first;  }
    public T2 getSecond ( ) {  return second;  }
    …
}
```

Defining the ordered pair class

```
public class Pair<T1,T2> {
    private T1  first;
    private T2  second;
    …
    public boolean equals ( Object other ) {
        if (other==null)                                return false;
        if (getClass() != other.getClass())             return false;
        Pair<T1,T2>  o = (Pair<T1,T2>)other;
        return first.equals(o.first) && second.equals(o.second);
    }
    …
}
```



A Primitive Type Cannot be Plugged in for a Type Parameter!!!

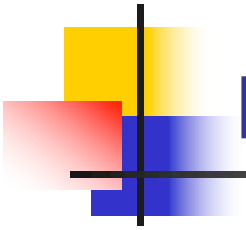
- The type plugged in for a type parameter must always be a reference type:
  - It cannot be a primitive type such as **`int`**, **`double`**, or **`char`**
  - However, now that Java has automatic boxing, this is not a big restriction.
  - **Note:** Reference types can include arrays.

Using Generic Classes and Automatic Boxing

Display 14.7 Using Our Ordered Pair Class and Automatic Boxing

```
1  import java.util.Scanner;

2  public class GenericPairDemo2
3  {
4     public static void main(String[] args)
5     {
6          Pair<Integer> secretPair =
7               new Pair<Integer>(42, 24);
8
9          Scanner keyboard = new Scanner(System.in);
10         System.out.println("Enter two numbers:");
11         int n1 = keyboard.nextInt();
12         int n2 = keyboard.nextInt();
13         Pair<Integer> inputPair =
14             new Pair<Integer>(n1, n2);

15         if (inputPair.equals(secretPair))
16         {
17             System.out.println("You guessed the secret numbers");
18             System.out.println("in the correct order!");
19         }
20         else
21         {
22             System.out.println("You guessed incorrectly.");
23             System.out.println("You guessed");
24             System.out.println(inputPair);
25             System.out.println("The secret numbers are");
26             System.out.println(secretPair);
27         }
28    }
29 }
```

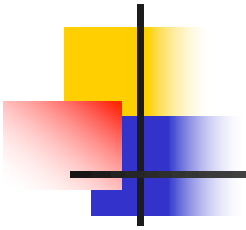
Automatic boxing allows you to use an `int` *argument for an* `Integer` *parameter.*



Limitations on Type Parameter Usage

- Within the definition of a parameterized class definition, there are places where an ordinary class name would be allowed, but a type parameter is not allowed.
- In particular, the type parameter cannot be used in simple expressions using new to create a new object
  - For instance, the type parameter cannot be used as a constructor name or like a constructor:

```
T object = new T();
T[] a = new T[10];
```



Limitations on Generic Class Instantiation

- Arrays such as the following are illegal:

```
Pair<String>[] a =
   new Pair<String>[10];
```

- Although this is a reasonable thing to want to do, it is not allowed given the way that Java implements generic classes