

Implementing Static Features

Based on slides by Prof. Burton Ma

Motivation

- ▶ You want to produce a software product that is to be used in many different countries
- ▶ Many different systems of measurement; for example
 - ▶ Distance: metre/kilometre versus yard/mile
 - ▶ Volume: teaspoon/tablespoon/cup versus millilitre/litre
 - ▶ Force: Newton versus pound-force
 - ▶ Currency: CAD versus USD versus EUR

Errors in Converting Units

- ▶ Errors in converting units can have catastrophic consequences
 - ▶ <http://lamar.colostate.edu/~hillger/unit-mixups.html>

Designing a Class to Convert Distances

- ▶ Design a class to convert between kilometres and miles
- ▶ What attributes are needed?
 - ▶ Number of kilometres per mile
 - ▶ Note: the number of kilometres in a mile never changes; it is genuinely a constant value
 - ▶ Attributes that are constant have all uppercase names

DistanceUtility
KILOMETRES_PER_MILE : double

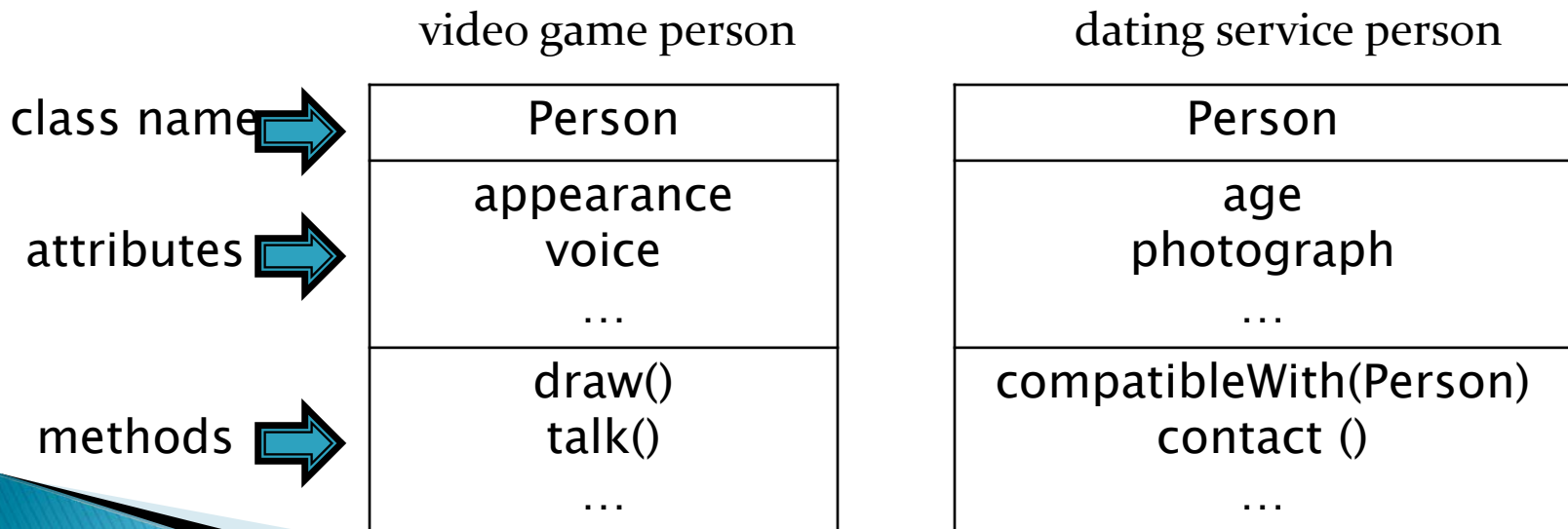
attribute type

Review: Java Class

- ▶ A class is a model of a thing or concept
- ▶ In Java, a class is the blueprint for creating objects
 - ▶ Attributes
 - ▶ The structure of an object; its components and the information (data) contained by the object
 - ▶ Methods
 - ▶ The behaviour of an object; what an object can do

Designing a Class

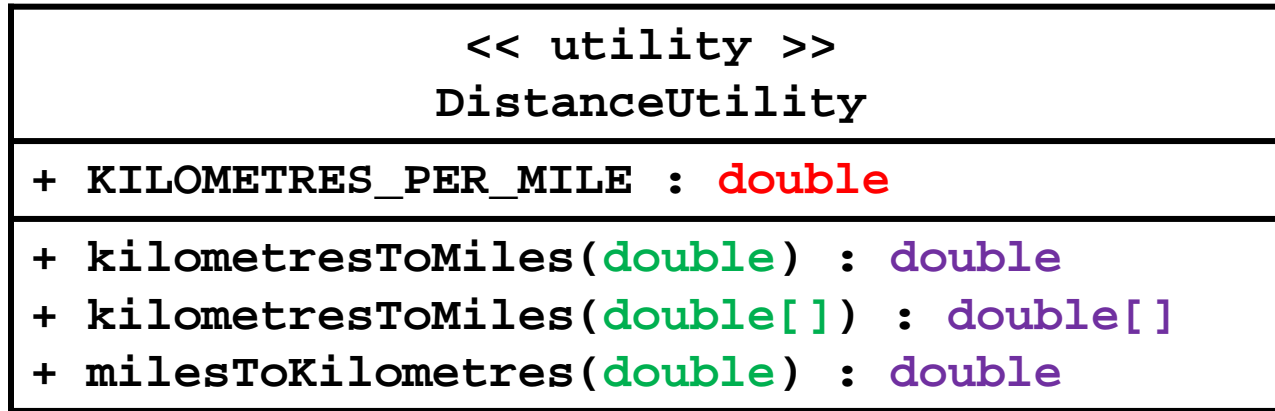
- ▶ To decide what attributes and methods a class must provide, you need to understand the problem you are trying to solve
 - ▶ The attributes and methods you provide depends entirely on the requirements of the problem



Utilities

- ▶ In Java, a *utility* class is a class having only static attributes and static methods
- ▶ Uses:
 - ▶ Group related methods on primitive values or arrays
`java.lang.Math` or `java.util.Arrays`
 - ▶ Group static methods for objects that implement an interface
`java.util.Collections`
 - ▶ Group static methods on a `final` class
 - ▶ More on this when we talk about inheritance

UML Class Diagram for Utilities



- ▶ Class name preceded by `<< utility >>`
- ▶ `+` means public (`-` means private)
- ▶ Attributes: `type`
- ▶ Methods: `parameters` and `return type`

General Class Structure

```
// any needed package statement
// any needed import statements

public class SomeName
{
    // the attribute section

    // the constructor section

    // the method section
}
```

Version 1

```
public class DistanceUtility
{
    // attributes
    public static final
        double KILOMETRES_PER_MILE = 1.609344;
}
```

Attributes

```
public static final
    double KILOMETRES_PER_MILE = 1.609344;
```

- ▶ An attribute is a member that holds data
- ▶ A constant attribute is usually declared by specifying

1. modifiers

1. access modifier

`public`

2. static modifier

`static`

3. final modifier

`final`

2. type

`double`

3. name

`KILOMETRES_PER_MILE`

4. value

`1.609344`

Attributes

- ▶ Attribute names must be unique in a class
- ▶ The scope of an attribute is the entire class
- ▶ `public` attributes often called “fields” [API]

public Attributes

- ▶ A **public** attribute is visible to all clients

```
public class NothingToHide
{
    public int x;    // always positive
}
```

```
// client of NothingToHide
NothingToHide h = new NothingToHide();
h.x = 100;
```

- ▶ **public** attributes break encapsulation
 - ▶ A `NothingToHide` object has no control over the value of `x`
 - ▶ Clients can put a `NothingToHide` object into an invalid state

```
h.x = -500; // x not positive
```

public Attributes

- ▶ A **public** attribute makes a class brittle in the face of change

```
public class NothingToHide
{
    private int x;  // always positive
}

// existing client of NothingToHide
NothingToHide h = new NothingToHide();
h.x = 100;  // no longer compiles
```

- ▶ **public** attributes are hard to change
 - ▶ They are part of the class API
 - ▶ Changing access or type will break existing client code

`public` Attributes

- ▶ Avoid `public` attributes in production code
 - ▶ Except when you want to expose constant value types

static Attributes

- ▶ An attribute that is `static` is a per-class member
 - ▶ Only one copy of the attribute, and the attribute is associated with the class
 - ▶ Every object created from a class declaring a static attribute shares the same copy of the attribute
 - ▶ Textbook uses the term *static variable*
 - ▶ Also commonly called *class variable*

static Attribute

- ▶ DistanceUtility u =
- ▶ new DistanceUtility();
- ▶ DistanceUtility v =
- ▶ new DistanceUtility();

u
v

KILOMETRES_PER_MILE

belongs to class →

no copy of
KILOMETRES_PER_MILE

???

???

64

client invocation

1000

1100

500

DistanceUtility class

1.609344

1000

DistanceUtility object

1100

DistanceUtility object

static Attribute Client Access

- ▶ A client should access a **public static** attribute without requiring an object
 - ▶ Use the class name followed by a period followed by the attribute name

```
// client of DistanceUtility  
double kmPerMi = DistanceUtility.KILOMETRES_PER_MILE;
```

static Attribute Client Access

- ▶ It is legal, *but considered bad form*, to access a `public static` attribute using an object

```
// client of DistanceUtility; avoid doing this
DistanceUtility u = new DistanceUtility();
double kmPerMi = u.KILOMETRES_PER_MILE;
```

new DistanceUtility Objects

- ▶ Our `DistanceUtility` API does not expose a constructor

but

```
DistanceUtility u = new DistanceUtility();
```

is legal

- ▶ If you do not define any constructors, Java will generate a default no-argument constructor for you

Preventing Instantiation

- ▶ Our `DistanceUtility` API exposes only **static** constants (and methods later on)
 - ▶ Its state is constant
- ▶ There is no benefit in instantiating a `DistanceUtility` object
 - ▶ A client can access the constants (and methods) without creating a `DistanceUtility` object

```
double kmPerMi = DistanceUtility.KILOMETRES_PER_MILE;
```

- ▶ Can prevent instantiation by declaring a **private** constructor

Version 2 (prevent instantiation)

```
public class DistanceUtility
```

```
{
```

```
    // attributes
```

```
    public static final double KILOMETRES_PER_MILE = 1.609344;
```

```
    // constructors
```

```
    // suppress default ctor for non-instantiation
```

```
    private DistanceUtility()
```

```
    {}
```

```
}
```

private

- ▶ **private** attributes, constructors, and methods cannot be accessed by clients
 - ▶ they are not part of the class API
- ▶ **private** attributes, constructors, and methods are accessible only inside the scope of the class
- ▶ A class with only **private** constructors indicates to clients that they cannot use **new** to create instances of the class

`final` Attributes

- ▶ An attribute (or variable) that is `final` can only be assigned to once
 - ▶ `public static final` attributes are typically assigned when they are declared

```
public static final double  
    KILOMETRES_PER_MILE = 1.609344;
```

- ▶ `public static final` attributes are intended to be constant values that are a meaningful part of the abstraction provided by the class

final Attributes of Primitive Types

- ▶ **final** attributes of primitive types are

```
public class AlsoNothingToHide
{
    public static final int x = 100;
}
```

```
// client of AlsoNothingToHide
AlsoNothingToHide.x = 88; // will not compile;
                          // attribute is final and
                          // previously assigned
```

final Attributes of Immutable Types

- ▶ **final** attributes of immutable types are constant

```
public class StillNothingToHide
{
    public static final String x = "peek-a-boo";
}
```

```
// client of StillNothingToHide
StillNothingToHide.x = "i-see-you";
                        // will not compile;
                        // attribute is final and
                        // previously assigned
```

- ▶ Also, **String** is immutable
 - ▶ It has no methods to change its contents

`final` Attributes

- ▶ Avoid using mutable types as `public` constants.

final Attributes of Mutable Types

- ▶ **final** attributes of mutable types are not logically constant; their state can be changed

```
public class LastNothingToHide
{
    public static final ArrayList<Integer> x =
        new ArrayList<Integer>();
}
```

```
// client of LastNothingToHide
ArrayList<Integer> y = new ArrayList<Integer>();
LastNothingToHide.x = y;    // will not compile;
                             // attribute is final and
                             // previously assigned

LastNothingToHide.x.add( 10000 );
                             // works!
```

Version 3 (with methods)

```
public class DistanceUtility
```

```
{
```

```
    public static final double KILOMETRES_PER_MILE = 1.609344;
```

```
    private DistanceUtility()
```

```
    {}
```

```
    // methods
```

```
    public static double kilometresToMiles(double km)
```

```
    {
```

```
        double result = km / KILOMETRES_PER_MILE;
```

```
        return result;
```

```
    }
```

```
}
```

Methods

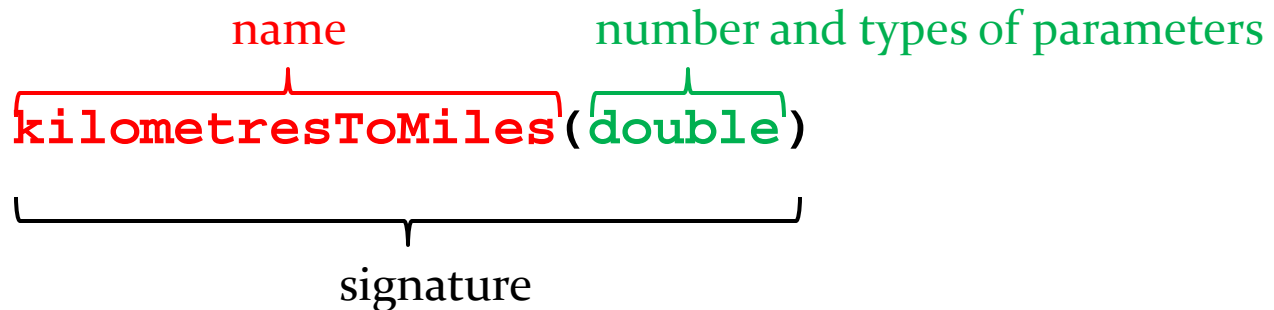
```
public static double kilometresToMiles(double km)
```

- ▶ A method is a member that performs an action
- ▶ A method has a signature (name + number and types of the parameters)

name number and types of parameters

kilometresToMiles(double)

signature

A diagram illustrating the components of a method signature. The signature 'kilometresToMiles(double)' is shown. A red bracket above 'kilometresToMiles' is labeled 'name'. A green bracket above '(double)' is labeled 'number and types of parameters'. A black bracket below the entire signature is labeled 'signature'.

- ▶ All method signatures in a class must be unique

Methods

```
public static double kilometresToMiles(double km)
```

- ▶ A method returns a typed value or `void`

`double`

- ▶ Use `return` to indicate the value to be returned

```
public static double kilometresToMiles(double km)
{
    double result = km / KILOMETRES_PER_MILE;
    return result;
}
```

Parameters

- ▶ Sometimes called *formal parameters*
- ▶ For a method, the parameter names must be unique
- ▶ The scope of a parameter is the body of the method

static Methods

- ▶ A method that is **static** is a per-class member
 - ▶ Client does not need an object to invoke the method
 - ▶ Client uses the class name to access the method

```
double miles = DistanceUtility.kilometresToMiles(100.0);
```

- ▶ **static** methods are also called *class methods*
- ▶ A **static** method can only use **static** attributes of the class

Invoking Methods

- ▶ A client invokes a method by passing arguments to the method
 - ▶ The types of the arguments must be compatible with the types of parameters in the method signature
 - ▶ The values of the arguments must satisfy the preconditions of the method contract

```
double kilometres = 100.0;  
double miles = 0.0;  
miles = DistanceUtility.kilometresToMiles(kilometres);
```

arguments

kilometres

miles

84

100.0

0.0

Arguments vs. Parameters

- ▶ Arguments are passed in a method call
- ▶ Parameters are accessed within the method

```
int x = 8;
int y = 10;
int max = Foo.max(x, y); // x and y are arguments

public static int max(int a, int b) // in class Foo
{
    return a > b ? a: b; // a and b are parameters
                        // where a=x and b=y
}
```

Attribute Shadowing

- ▶ What if a parameter and attribute share the same name?
- ▶ Parameter takes priority

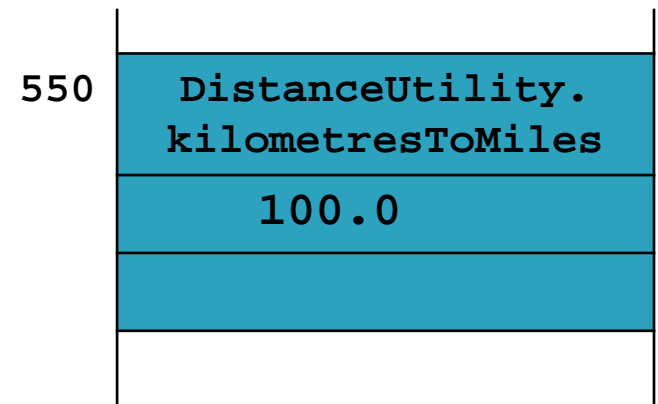
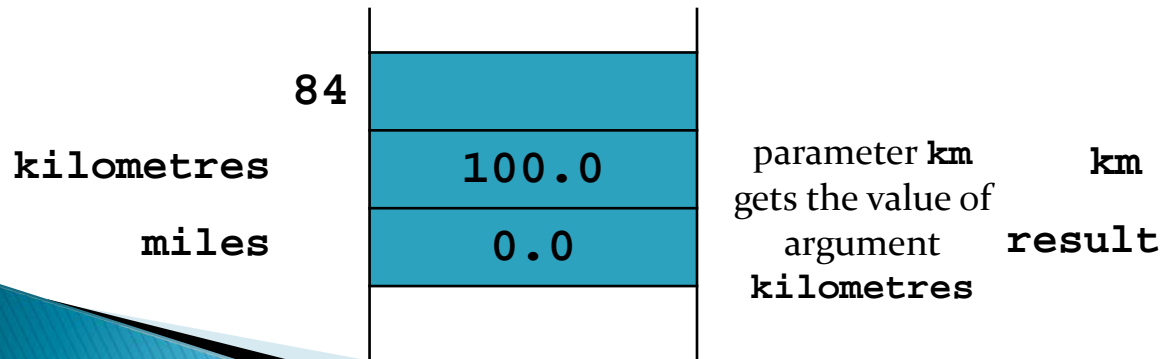
```
public class Foo
{
    static int count; // attribute
    ...
    public static void increment(int count)
    {
        count = count + count; // changes parameter, not attribute
        Foo.count = Foo.count + count; // changes attribute
        // this.count if Foo were not a Utility class
    }
}
```

Pass-by-value with Primitive Types

- ▶ An invoked method runs in its own area of memory that contains storage for its parameters
- ▶ Each parameter is initialized with the value of its corresponding argument

```
miles =  
    DistanceUtility.kilometresToMiles(  
        kilometres);
```

```
public static double  
    kilometresToMiles(double km)
```

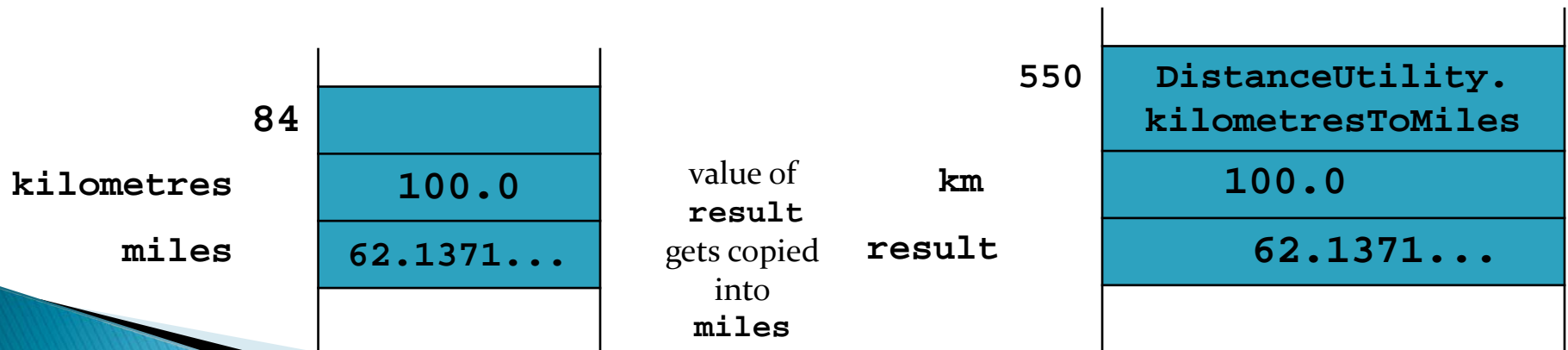


Pass-by-value with Primitive Types

- ▶ The method body runs and the return value is computed
- ▶ The return value is then copied back to the caller

```
miles =  
    DistanceUtility.kilometresToMiles(  
        kilometres);
```

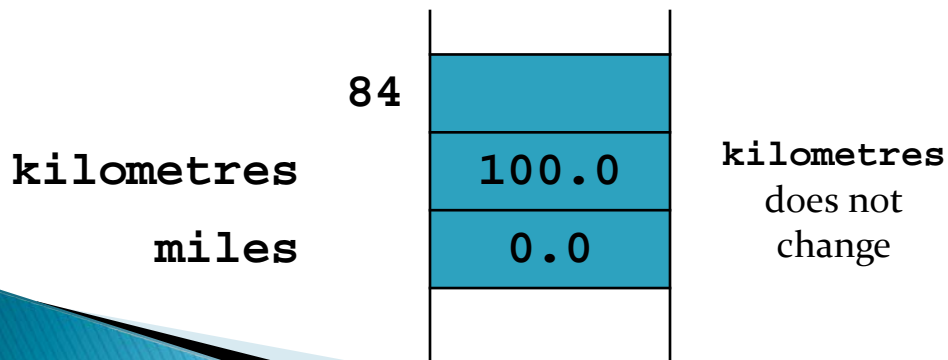
```
public static double  
    kilometresToMiles(double km)
```



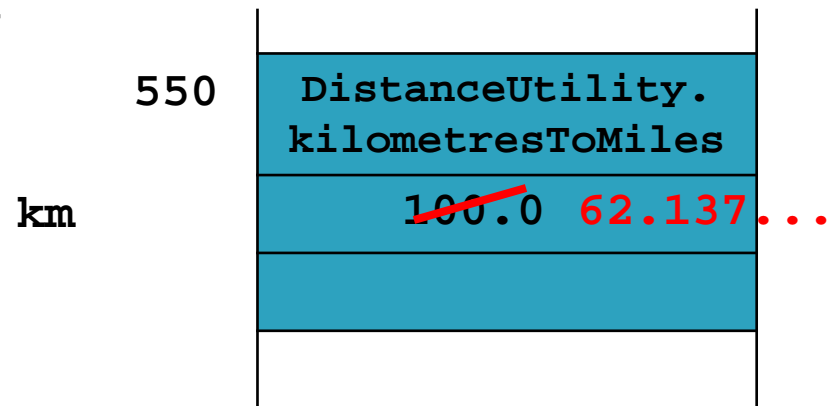
Pass-by-value with Primitive Types

- ▶ The argument `kilometres` and the parameter `km` have the same value but they are distinct variables
- ▶ When `DistanceUtility.kilometresToMiles()` changes the value of `km` the value of `kilometres` does not change

```
miles =  
    DistanceUtility.kilometresToMiles(  
        kilometres);
```



```
public static double  
    kilometresToMiles(double km){  
        km /= KILOMETRES_PER_MILE;  
        return km;  
    }
```



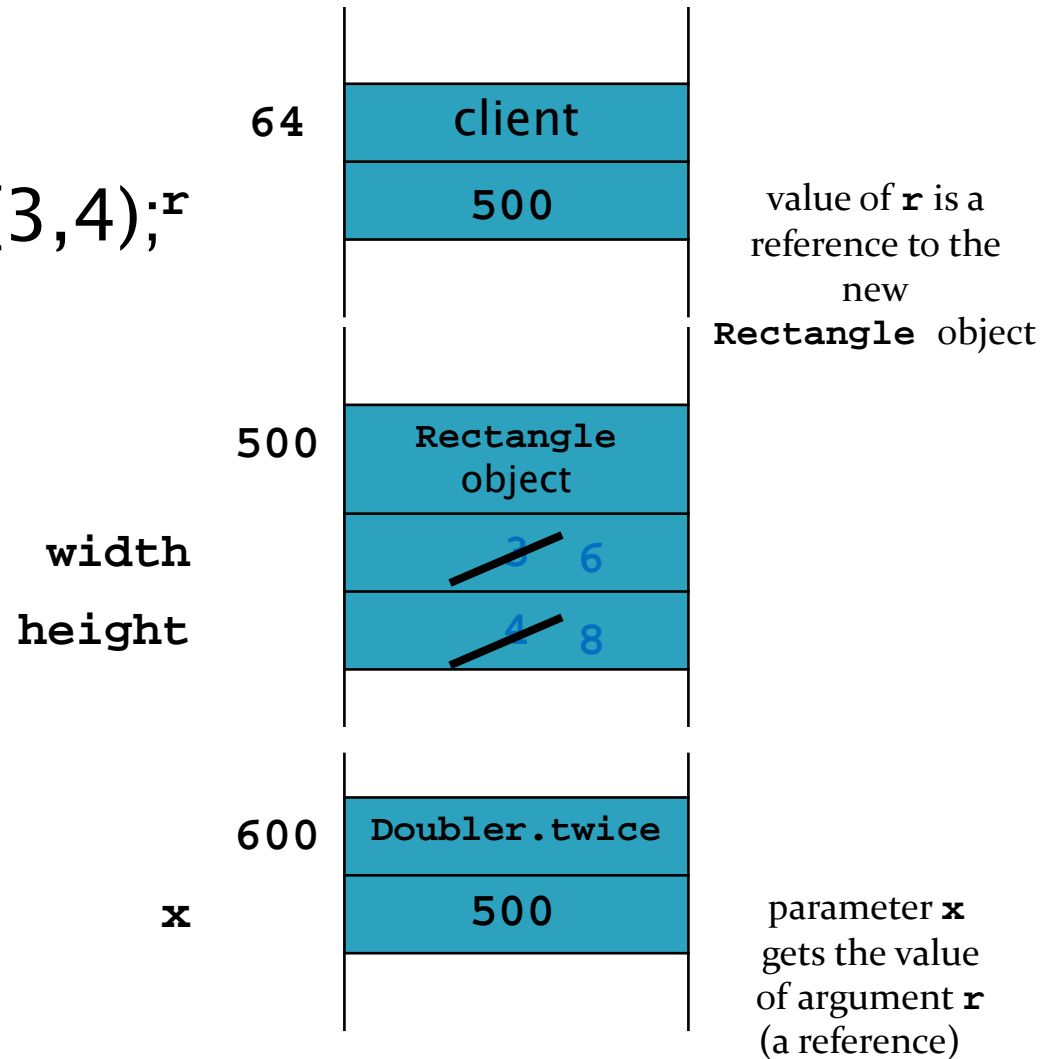
Pass-by-value with Reference Types

- ▶ Java uses pass-by-value for primitive and reference types

```
public class Doubler
{ // attributes and constructors not shown
    public static void twice(Rectangle x)
    {
        x.setWidth(2 * x.getWidth());
        x.setHeight(2 * x.getHeight());
    }
}
```


Pass-by-value with Reference Types

- ▶ `r = new Rectangle(3,4);`
`r`
- ▶ `Doubler.twice(r);`



Pass-by-value

- ▶ Java uses pass-by-value for primitive and reference types
 - ▶ An argument of primitive type cannot be changed by a method (e.g., no swapping of ints)
 - ▶ An argument of reference type can have its state changed by a method

Version 4 (Javadoc) 1

```
/**
 * The class DistanceUtility contains constants and
 * methods to convert between kilometres and miles.
 *
 * @author EECS1030
 */
public class DistanceUtility
{
    /**
     * The number of kilometres in a mile.
     */
    public static final double KILOMETRES_PER_MILE = 1.609344;
```

Version 4 (Javadoc) 2

```
/**
 * Converts distances in kilometres to miles.
 *
 * @param km The distance to convert. If km is
 *           negative then the returned distance is also negative.
 * @return Distance in miles.
 */
public static double kilometresToMiles(double km)
{
    double result = km / KILOMETRES_PER_MILE;
    return result;
}
```

Overloading `kilometresToMiles()`

- ▶ Suppose we want to provide a method to convert many values stored in an array from kilometres to miles
 - ▶ We can provide another method called `kilometresToMiles()` as long as the signature is different
- ▶ Providing multiple methods with the same name but different signatures is called *method overloading*
- ▶ The intent of overloading is to provide flexibility in the types of arguments that a client can use

Version 4 (overload a method)

```
public class DistanceUtility
{
    // attributes and constructors; see Version 2 or 2a ...

    // methods
    public static double kilometresToMiles(double km)
    { // see version 3}

    public static double[] kilometresToMiles(double[] km)
    {
        double[] miles = new double[km.length];
        for(int i = 0; i < km.length; i++)
        {
            miles[i] = kilometresToMiles(km[i]); // good!
        }
        return miles;
    }
}
```

Method Overloading

- ▶ Simple rule
 - ▶ A class can define multiple methods with the same name as long as the signatures are unique

```
// DistanceUtility examples
```

```
kilometresToMiles(double)
```

```
kilometresToMiles(double[])
```

```
// String examples
```

```
String()
```

```
String(char[] value)
```

```
String(char[] value, int offset, int count)
```

Overloading 1

- ▶ Everything other than the signature is ignored in determining a legal overload

```
// illegal; parameter names not part of signature  
// add this to DistanceUtility: legal or illegal?  
public static double kilometresToMiles(double kilos)
```


Overloading 2

```
// illegal; access modifier not part of signature  
// legal or illegal?  
private static double kilometresToMiles(double km)
```

Overloading 3

```
// illegal; static modifier not part of signature  
// legal or illegal?  
public double kilometresToMiles(double km)
```

Overloading 4

```
// illegal; return type not part of signature  
// legal or illegal?  
public static float kilometresToMiles(double km)
```

Overloading 5

```
// legal; parameter type is part of signature
// legal or illegal?
public static float kilometresToMiles(float km)
{
    // this works
    return (float)(km / KILOMETRES_PER_MILE);
}
```

Overloading 5a

```
// implemented in terms of kilometresToMiles(double)
//
public static float kilometresToMiles(float km)
{
    // but this might be better
    return (float) kilometresToMiles((double) km);
}
```

Selection of Overloaded Methods (i.e., Binding)

- ▶ Loosely speaking, the compiler will select the method that most closely matches the number and types of the arguments
 - ▶ “The rules that determine which overloading is selected are extremely complex. They take up thirty–three pages in the language specification [JLS, 15.12.1–3], and few programmers understand all of their subtleties.”
 - ▶ Effective Java, Second Edition, p 195.

Selection Examples

// from java.lang.Math

Math.abs(-5);	// Math.abs(int a)
Math.abs(-5f);	// Math.abs(float a)
Math.abs(-5.0);	// Math.abs(double a)

Math.max(1, 2);	// Math.max(int a, int b)
Math.max(1.0, 2.0);	// Math.max(double a, double b)
Math.max(1, 2.0);	// Math.max(double a, double b)



no exact match for `Math.max(int, double)`
but the compiler can convert `int` to `double`
to match `Math.max(double, double)`

Ambiguous Overloads

```
public class Ambiguous
{
    public static void f(int a, double b)
    {
        System.out.println("f int double");
    }
    public static void f(double a, int b)
    {
        System.out.println("f double int");
    }
    public static void main(String[] args)
    {
        f( 1, 2 ); // Compilation error:
                   // "reference to f is ambiguous, both methods
                   // f(int,double) and f(double,int) match"
    }
}
```


Confusing Overload

```
import java.util.*;

public class SetList
{
    public static void main(String[] args)
    {
        Set<Integer> set = new TreeSet<Integer>();
        List<Integer> list = new ArrayList<Integer>();
        // fill set and list with -3, -2, -1, 0, 1, 2
        for(int i = -3; i < 3; i++)
        {
            set.add(i); list.add(i);
        }
        System.out.println("before " + set + " " + list);
    }
}
```

[Effective Java, Second Edition, p 194]

Confusing Overload

```
// remove 0, 1, and 2?  
for(int i = 0; i < 3; i++)  
{  
    set.remove(i); list.remove(i);  
}  
System.out.println("after " + set + " " + list);  
}  
}
```

[Effective Java, Second Edition, p 194]

Confusing Overload Explained 1

before [-3, -2, -1, 0, 1, 2] [-3, -2, -1, 0, 1, 2]
after [-3, -2, -1] [-2, 0, 2]

- ▶ `set` and `list` are collections of `Integer`
 - ▶ Calls to `add` autobox their `int` argument

```
set.add(i); // autobox int i to get Integer  
list.add(i); // autobox int i to get Integer
```

- ▶ Calls to `TreeSet` `remove` also autobox their `int` argument

```
set.remove(i); // autobox int i to get Integer
```

Confusing Overload Explained 2

- ▶ However, `ArrayList` has an overloaded `remove` method

`remove(int index)`

Removes the element at the specified position in this list.

- ▶ Therefore, `list.remove(i)` matches the `int` version of `remove()` instead of the `Integer` version of `remove()`

```
list.remove(0); // [-3, -2, -1, 0, 1, 2]
list.remove(1); // [-2, -1, 0, 1, 2]
list.remove(2); // [-2, 0, 1, 2]
```

Interfaces and Generics as Parameters

- ▶ When specifying parameters for a method, one can use interfaces instead of classes **as long as the interface declares the required functionality**
 - This allows for passing of objects from any implementing class
 - Can allow for a type of collection (e.g., a Map) instead of an implementation (e.g., a TreeMap)



```
public static int freq(List<Long> list)
```

instead of



```
public static int freq(ArrayList<Long> list)
```

Interfaces and Generics as Parameters (2)

- ▶ If the parameterized type (e.g., the type of element in a collection) is not known (or does not matter), one can specify a generic type
 - Typically written as “E”, “T”, “K”, or “V”
 - Can also be used to describe the return type

```
public static <T> int freq(List<T> list)
```

- Declares <T> as the type in the generic method “freq”

```
public static <T> T getLargest(List<T> list)
```

- <T> declares “freq” as a generic method, and its return type is also an object of type T

Interfaces and Generics as Parameters (3)

- ▶ Sometimes, restrictions are required on the type (e.g., elements must be comparable)

```
public static <T extends Comparable<T>>  
    boolean smaller(List<T> list)
```

- T must implement Comparable directly

```
public static <T extends Comparable<? super T>>  
    boolean smaller(List<T> list)
```

- Allows T if it (or one of its superclasses) implements Comparable directly

What to do About Invalid Arguments

- ▶ As the author of a class, you have control over how your method is implemented
- ▶ What you cannot control is the value of the arguments that clients pass in
- ▶ A well written method will
 1. Specify any requirements the client must meet with the arguments it supplies → preconditions
 2. Validate the state of any arguments without preconditions and deal gracefully with invalid arguments → validation

Preconditions

- ▶ If a method specifies a precondition on one of its parameters, then it is the client's responsibility to make sure that the argument it supplies satisfies the precondition
 - ▶ If a precondition is not satisfied then the method can do anything (such as throw an exception, return an incorrect value, behave unpredictably, ...)
- ▶ For our method possible preconditions are:
 - ▶ `km` must not be null
 - ▶ `km.length > 0`
 - ▶ Note that the second precondition is more restrictive than the first

```
/**  
 * Converts distances in kilometres to miles for arrays.  
 * If an element of the array argument is negative the  
 * corresponding element of the returned array is also  
 * negative.  
 *  
 * @param km The distances to convert.  
 * @pre.      <code>km.length > 0</code>  
 * @return Distances in miles in an array with  
 *          <code>length == km.length</code>.  
 */  
public static double[] kilometresToMiles(double[] km)
```

Validation

- ▶ Alternatively, the class implementer can relax preconditions on the arguments and validate the arguments for correctness
- ▶ The implementer assumes the responsibility for dealing with invalid arguments
 - ▶ Must check, or *validate*, the arguments to confirm that they are valid
 - ▶ Invalid arguments must be accommodated in a way that allows the method to satisfy its postconditions
- ▶ In our example, a possible return value for a `null` array is a zero-length array

```
/**
 * Converts distances in kilometres to miles for arrays.
 * If an element of the array argument is negative the
 * corresponding element of the returned array is also
 * negative.
 *
 * @param km    The distances to convert.
 * @return      Distances in miles in an array with
 *               <code>length == km.length</code>. If the
 *               array argument is <code>null</code> then a
 *               zero-length array is returned.
 */
```

```
public static double[] kilometresToMiles(double[] km)
{
    double[] miles = null;
    if (km == null) {
        miles = new double[0];
    }
    else {
        miles = new double[km.length];
        for(int i = 0; i < km.length; i++) {
            miles[i] = kilometresToMiles(km[i]);
        }
    }
    return miles;
}
```

Loop Invariant

- ▶ A Boolean expression that is true (i.e., holds) at the beginning of every iteration of the loop
- ▶ Does not necessarily appear in code
- ▶ Typically derived for testing purposes and to prove correctness

Loop Invariant (2)

```
int pow = 1;
int i = 0;
while (i < exponent)
{
    pow = pow * base;
    i++;
}
return pow;
```

► Invariants:

- $i \geq 0$
- $i \leq \text{exponent}$
- $\text{pow} == \text{base}^i$

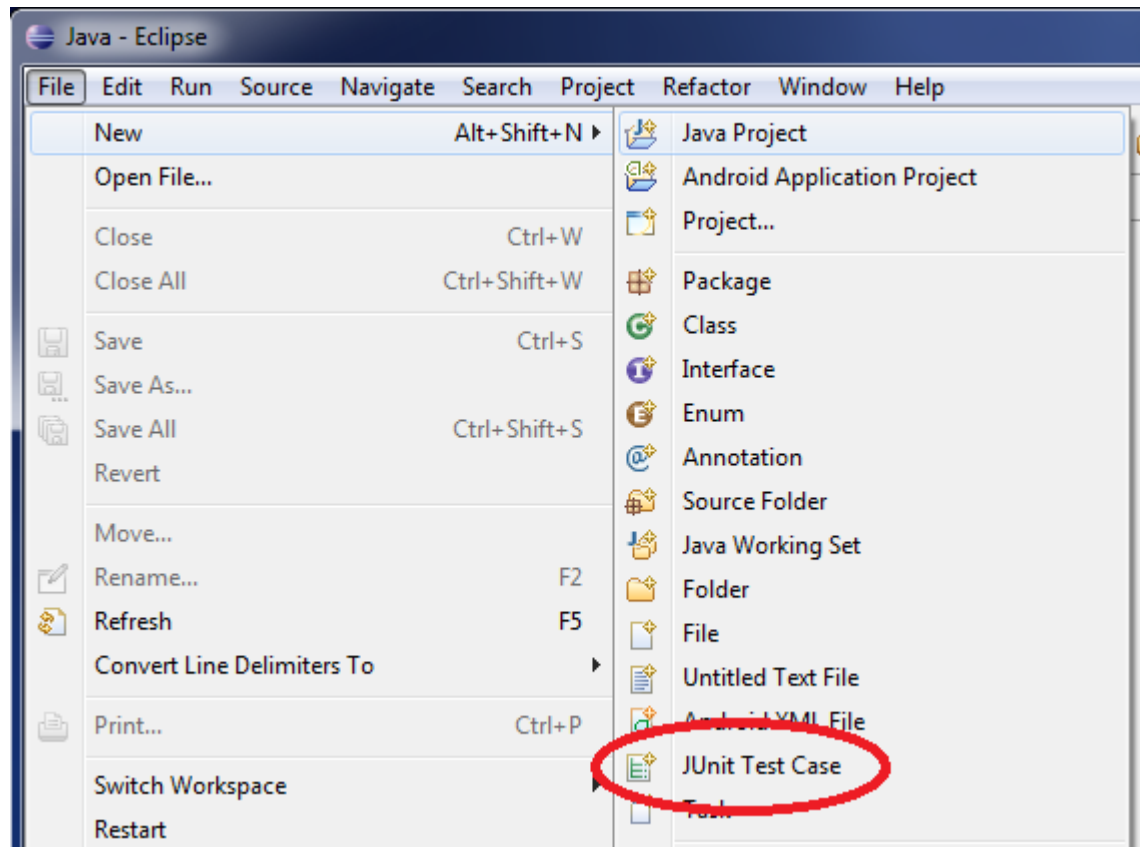
Testing

- ▶ Imperative to test utility (and all) classes for correctness
- ▶ Compare calculated output with expected output
 - Identical result → test passed
 - Different result → test failed
- ▶ Testing requires multiple test cases to ensure correct operation under various condition with various inputs
- ▶ Example: Test kilometresToMiles method

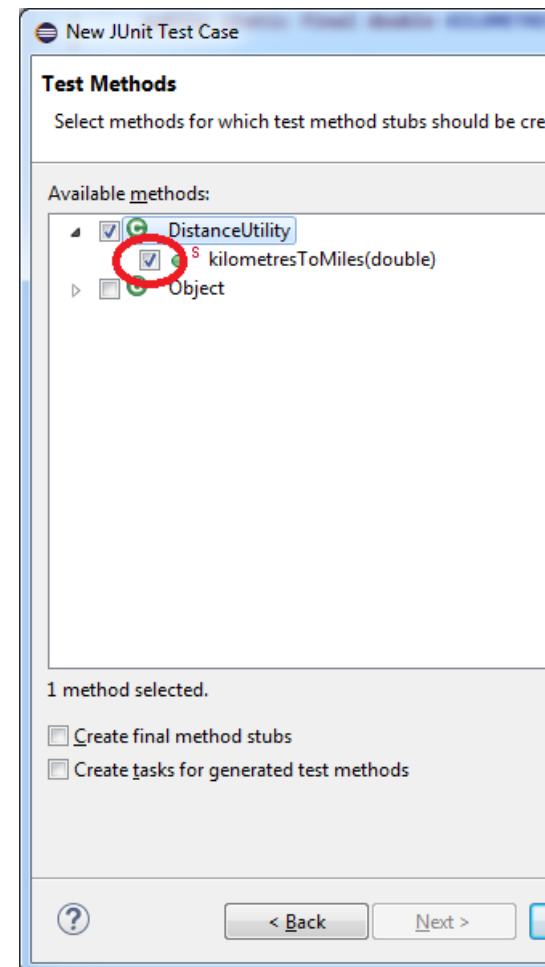
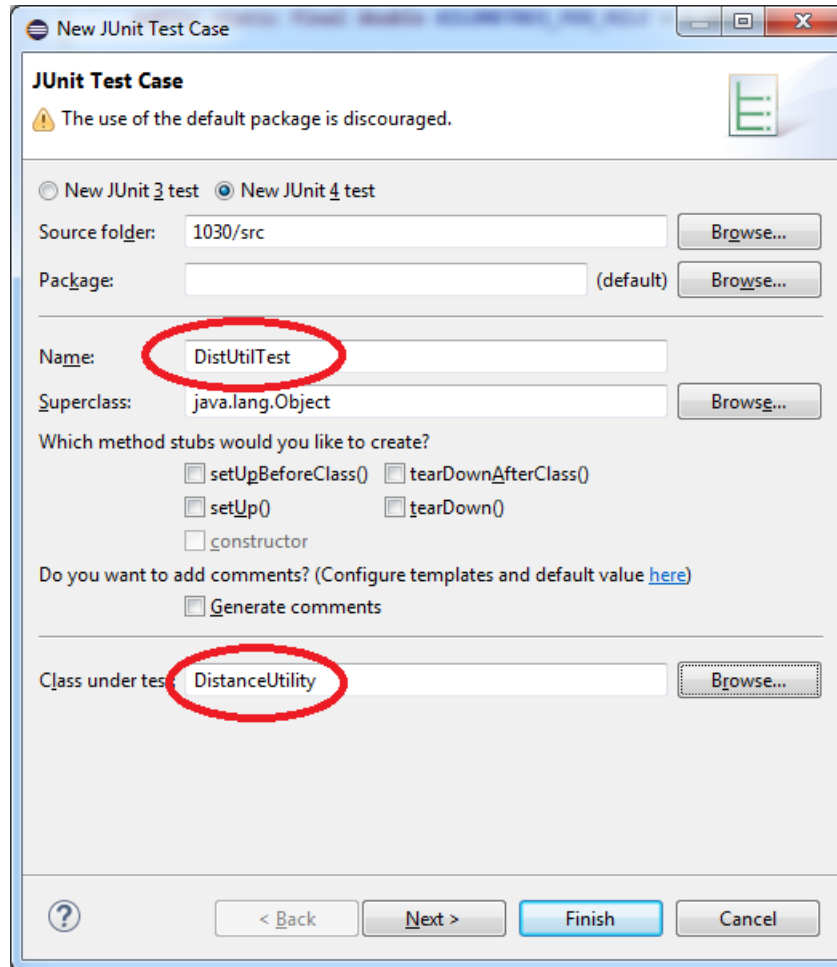
Testing (Testing Class)

```
public class DistUtilTester
{
    public static void main(String[] args)
    {
        double input = 2;
        double expected = 1.24274238; // used calculator as oracle
        double actual = DistanceUtility.kilometresToMiles(input);
        double epsilon = 0.000001;
        if (Math.abs(actual - expected) < epsilon)
        {
            System.out.println("passed");
        }
        else
        {
            System.out.println("failed");
        }
    }
}
```

Testing (JUnit in Eclipse)



Testing (JUnit in Eclipse) (2)



Testing (JUnit in Eclipse) (3)

```
@Test
public void testKilometresToMiles()
{
    double input = 2;
    double expected = 1.24274238; // calculator as oracle
    double actual = DistanceUtility.kilometresToMiles(input);
    double epsilon = 0.000001;
    assertEquals("Actual and expected values exceed epsilon!",
        expected, actual, epsilon);
}
```

Testing (JUnit in Eclipse) (4)

