Parameters

Lecture 5 - A
Object-Oriented Programming

Agenda

- Why Parameters?
- Message Specification
- Smart Objects
- Sending and Receiving Parameters
- Actual and Formal Parameter
- CircleCalculator Example
- Formal Parameter
- Actual and Formal Parameters
- Signature
- References in Java
- Parameters and Association
- The Receiver & The Sender syntax
- Types of Methods Revisited
- Return Types
- Accessor and Mutator Methods
- Readings

Why Parameters (1)

- The world of objects is all about cooperating objects.
- In order to cooperate the objects have to send messages to each other.
- To communicate, objects send messages to each other.
- How do we make messages specific?
- Answer: *Parameters*

Why Parameters (2)

- Objects have two types of relationship
  - Containment
  - Association
- Objects contain other objects and have knowledge of these objects.
- However, the contained objects does not have the knowledge of its container.
- How do we provide such knowledge?
- Answer: *Parameters*

Making Messages Specific

- `OOP_Car` needs a paint job!
- Let's say we want to give `OOP_Car` the capability of being painted different colors
- One solution:
  - add a method to `OOP_Car` for each color we want to paint with

```
public void setRed();
public void setBlue();
public void setTeal();
public void setMauve();
...
```

- Not very elegant to write all these methods
- Much more efficient to write *one* setColor method in class `OOP_Car` in which we could *specify the color* that we want to paint with

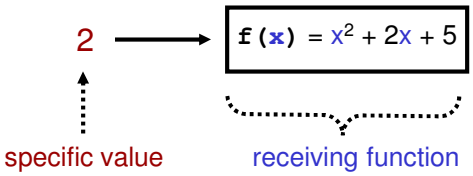
Making Smart Objects

- `OOP_Car` is going for a drive!
- It's time to associate `OOP_Car` with the `City` it's driving in
  - This way it can call methods on class `City` to ask where schools and parks are located
- Remember, `City` *contains* `OOP_Car`
  - `City` has an instance variable of class `OOP_Car`
  - This means `City` can call methods on this instance
- But containment relationship is not symmetric
  - `OOP_Car` can't automatically call methods on `City` (it doesn't know about the city)
- So how can we enable `OOP_Car` to know its container, `City`, so it can call methods on `City`?
- Need to associate a `City` instance with a `OOP_Car` instance in order for a `OOP_Car` to call methods on `City`

How do we associate two objects?

Sending and Receiving Parameters

- Mathematical Analogy
  - a function in math is like a method that *receives* one or more parameters and computes a result
  - "*send*" the function a specific value of the parameter



Actual and Formal Parameter

- **x** is a *Formal* parameter
  - *formal* parameters are "dummy" variables that represent the type of object that will be passed in when the method is called
  - have no value of their own; take on value of parameters passed in when method is called
  - placeholders, like x in $x^2 + 2x + 5$
- **2** is an *Actual* parameter
  - *actual* parameters are "actual" instances sent when calling method
  - sender passes a specific value/instance with method call to receiver

CircleCalculator

```
public class CircleCalculator{

  private final double PI ;
  private double radius;

  public CircleCalculator() //Constructor of the CircleCalculator Class
  {
    PI = 3.14159;
    radius = 40.0;
  }

  public void printCircumfarence()
  {
    System.out.println(2*PI*radius);
    return;
  }

  public void printArea()
  {
    System.out.println(PI*radius*radius);
    return;
  }
}
```

radius is fixed in this class

Lets use parameters and make radius more specific

Improved CircleCalculator

```
public class CircleCalculator{

  private final double PI ;

  public CircleCalculator() //Constructor of the
   CircleCalculator Class
  {
    PI = 3.14159;
  }

  public void printCircumfarence(int radius)
  {
    System.out.println(2*PI*radius);
    return;
  }

  public void printArea(int radius)
  {
    System.out.println(PI*radius*radius);
    return;
  }
}
```

radius is used as a parameter

Formal Parameter

```
public void printCircumfarence(int radius)
  {
   System.out.println(2*PI*radius);
    return;
  }

public void printArea(int radius)
  {
    System.out.println(PI*radius*radius);
    return;
  }
```

Formal Parameters

- **The parameter "radius" is the formal parameter of these two methods. It is given a type "int".**
- **In the method it will behave as a local variable with supplied values.**

Using `CircleCalculator`

```
CircleCalculator circleCalculator = new CircleCalculator();
```

Instantiating CircleCalculator Object

```
circleCalculator.printCircumfarence(45);
```

CircleCalculator Object | Sending Message | Actual Parameter

Actual and Formal Parameters

- *One-to-one* correspondence between formal and actual parameters
  - order, type (class) of instances, and number of parameters sent *must match* order, type, and number declared in method!

```
circleCalculator.printCircumfarence(45);
```
  - sends one parameter of type `int`
  - which matches:

```
public void printCircumfarence(int radius)
```
  - expects one parameter of type `int`

- Name of formal parameter does not have to be the same as the corresponding actual parameter
  - receiver *cannot know* what specific instances will be passed to it, yet it *must know* how to refer to it
    - receiver uses a dummy name to represent the parameter
  - sender may send different actual parameters
    - Any value falling in the range of int could be sent

Signature

A method's *signature* is composed of its…

- **Identifier (name)**
- **Classes (types) of its parameters**
- **order of its parameters**

- Like a person's signature, a method's signature must be unique.

- Creating two methods in one class with the same signature will cause an error

More on References in Java

```
CircleCalculator circleCalculator = new CircleCalculator();
```

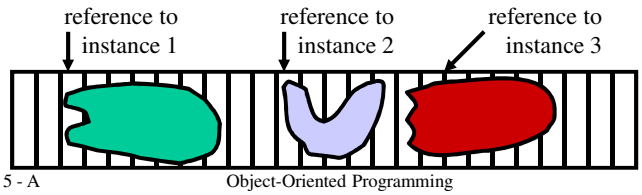
What is circleCalculator?

circleCalculator is not an object itself. It is a reference to the location where the object is created by Java.

Java never allows allow access to objects directly. It only gives a reference to access the instantiated objects.

References in Java

- A reference is just a pointer to a location in memory
  - a version of an address people can understand
  - it's easier to keep track of a name than some weird …thing (like **`0xeff8a9f4`**)
  - holds a memory address where instance is stored
  - Java also has **primitives**, which are not objects and do not have pointers



Parameters and Association

- Let's use what we've learned about parameters to enable a `OOP_Car` to *know about* its `City` -- called *association*
  - `OOP_Car` can store a reference to its `City` so that it can send messages to it
- Usually associations are done in the constructor
  - don't forget that constructors are methods, too
  - they can receive parameters just like any other method

Syntax: The *Receiver*

```
public class OOP_Car {
  private City _city;

  public OOP_Car(City myCity) {
    _city = myCity;

  }
}
```

So whoever instantiates a `OOP_Car` is expected to pass in a `City` to the `OOP_Car's` constructor for the `OOP_Car` to be associated with.

The instance `City` will be referenced by the formal parameter name `myCity` whose value is then assigned to the instance variable `_city`. Now the `OOP_Car` can call any of `City`'s methods on `_city`.

Syntax: The *Sender*

```
public class City {

  private OOP_Car _15mobile;

  public City() {
    _15mobile = new OOP_Car(this);
  }

}
```

In this case, the "whoever" mentioned on the previous slide is the **City** itself. It passes the **OOP_Car** a reference to itself by using the reserved word **this**.

Now **_15mobile** is associated with "this" instance of **City** .

Syntax for the Receiver

- Syntax for class **OOP_Car**

```
public class OOP_Car {
  private City _city;

  // _city is a variable representing the
  // City that this OOP_Car is driving in

  // OOP_Car constructor
  public OOP_Car(City myCity) {
     _city = myCity;
  }
}
```

- standard constructor that *receives* a parameter
- assigns the parameter passed in, **myCity**, to the instance variable **_city**
- **OOP_Car** now *knows about* **myCity**

Syntax for the Sender

- Syntax for class `City`

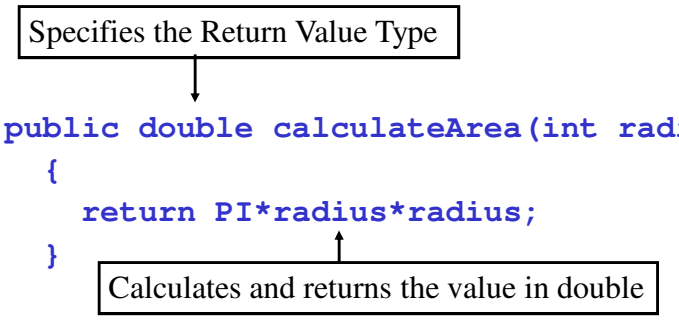
```
_15mobile = new OOP_Car(this);
```

- Remember **`this`**
  - shorthand for "this instance", i.e., instance where execution is currently taking place
- Constructor for `OOP_Car` needs a `City`
  - we need to pass an instance of the `City` class to the constructor for `OOP_Car`
  - where can we find an instance of `City`?
  - since we're in the `City` class constructor (i.e., execution is taking place inside the `City` class) we can use `this` as our instance of `City`

Types of Methods Revisited

- *An object's method or capabilities*
  - *constructors*: establish initial state of object's properties
  - *commands*: change object's properties
  - *queries*: provide answers based on object's properties
- We have seen the methods of first two types.
- Let's what we need for the methods of the third type.

Return Types



- We are passing a formal parameter of type int.
- The method returns the area of the circle of type double.

Return Types (2)

```
double area; //Declaring a local variable namely area
CircleCalculator circleCalculator = new
  CircleCalculator();
//Instantiating an object of CircleCalculator class
area = circleCalculator.calculateArea(40);
```

Must of a double type

Must of an int type

- When we call the method `calculateArea()` we need a local variable to get the value back from this method.
- The type of the local variable must match the return type of the method.

Accessor and Mutator Methods

- *Accessor* and *Mutator* methods are simple helper methods used to "get" or "set" some property of an object
- These methods are very simple but illustrate the concepts of parameters and return types
- Mutator Methods:
  - "set" methods
  - use parameters but generally do not return anything (return `void`)
  - exist primarily to change the value of a particular private instance variable (property) *in a safe way*

Accessor and Mutator Methods

- Accessor Methods:
  - "get" methods
  - use return types but generally do not take parameters
  - exist primarily to return information to the calling method
  - names usually start with *get* (`getColor`, `getSize`, `getCity`)
  - Examples, respectively:
    - method that returns the color of an object
    - method that returns the size of an object
    - method that returns the city that a `CP07Mobile` drives in
- Accessors and Mutators usually occur in pairs

Readings

Book Name: Object Oriented Programming in Java – A Graphical Approach

Author: Kathryn E. Sanders & Andries van Dam

Content: Chapter 2

Acknowledgements

- While preparing this course I have greatly benefited from the material developed by the following people:
  - Andy Van Dam (Brown University)
  - Mark Sheldon (Wellesley College)
  - Robert Sedgewick and Kevin Wayne (Princeton University)
  - Mark Guzdial and Barbara Ericsson (Georgia Tech)
  - Richard Halterman (Southern Adventist University)