**BIOE60010 Software Engineering for Bioengineers Week 2**

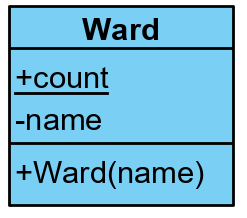
**Encapsulation, Inheritance. Introduction to Java GUIs: AWT, Git – managing changes**

There are four pillars of Object Oriented Programming – Encapsulation, Inheritance, Abstraction, Polymorphism. This week we will look at two of them. Before that, we need to understand class methods and class fields to help with our examples.

1. **Class Fields and Class Methods**

The keyword ‘static’ in front of a class’s field means that the variable is one and the same variable for every object that is an instance of that class, and it is therefore associated with a class rather than an object. They are known as class fields. Because there is only one memory location for a class field, if you change the value of the field via one instance of the class, the change is seen by all instances.

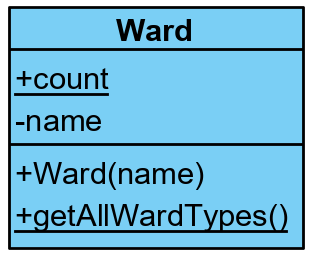
For example, consider a class ‘Ward’ in a hospital database.

**public class Ward {  
 public static int *count* = 0;  
 private String name;  
 public Ward(String name) {  
 *count*++;  
 this.name = name;  
 }  
}**

Class fields exist even if we haven’t instantiated any objects of that class. We could access ‘count’ therefore either via an instantiated object of that class eg **ward1.count** or by using the name of the class **Ward.*count*.**

Methods can also be static and accessed using the class name without instantiating an object, but they therefore cannot use any fields or methods that are not static. They tend to be used more for utility functions associated with that class.

For example,

**public class Ward {**

**public static int *count* = 0;  
 private String name;  
 public Ward(String name) {**

**count++;  
 this.name = name;  
 }  
 public static String[] getAllWardTypes() {  
 return new String[] {"General", "Maternity"};  
 }**

**}**

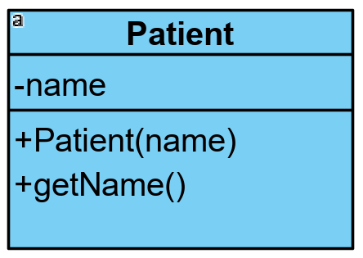
which could be used as follows:-

**String[] wardTypes**=**Ward**.***getAllWardTypes***();

1. **Encapsulation**

Consider the following definition of a Patient information class

**public class Patient {**

** *// fields*  
private String name;**

***// methods* public Patient(String name){**

**this.name = name; }  
public String getName(){  
 return name;  
 }  
}**

**Access control**

The process of controlling the access that external code has to a class’s fields and methods to keep access to the minimum required is known as **encapsulation.** The following access modifiers aid encapsulation:-

public: any code can access these fields and methods

private: only code within the class itself can access these fields and methods

protected: only code within the class itself or a class inherited from this class can access these fields and methods.

If there is no modifier, the field or method is ‘package-private’, which means that any code within the class’s package can access the field or method.

For example, if we instantiated an object of class Patient from within class Main:-

**Patient p = new Patient("John Smith");**

**p.getName(); // Valid: getName is public so can be called**

**String nm=p.name; // Error: name is private and so cannot be accessed**

**Getters and Setters**

Another aspect of encapsulation is to use public methods to make sure that the values put into fields are valid. Suppose there was a field of class String in the Patient class to hold the patient’s phone number, and a class called **TextSystem** that handles calling and messaging to bed phones via static methods – we could use a method known as a ‘setter’ to make sure that the number being set in a Patient object was valid, and a method called a ‘getter’ to get its current value:-

**private String mobileNum=null;**

**public void setMobileNum(String mobileNum) {**

**if (!TextSystem.checkValidNumber(mobileNum))**

**throw new Exception(“Invalid phone number”);**

**this.mobileNum=MobileNum;  
}**

**public String getmobileNum() {  
 return mobileNum;  
}**

1. **Inheritance**

A class can *inherit* the fields and methods of another class and then add its own more specific fields and methods. The inheriting class is called the subclass, and is more a specialised version of the inherited class. The inherited class is called the superclass of the relationship. The Java terminology is that the subclass ‘extends’ the superclass. The superclass in a chain of inheritance that does not inherit from any other is called the ‘base’ class. Inheritance can prevent duplication of code, as well as create a conceptual structure to better help understand a codebase.

Note: all Java base classes implicitly inherit the class ‘Object’.

For example, here is a UML class diagram for a hospital messaging system which shows the relationships between a class that represents a Patient and a class that represents a Doctor. On a class diagram, the open arrows can be read as ‘is a more specialized’ – ie they indicate inheritance. A +sign before a method or field indicates public, a -sign indicates private, and a #sign indicates protected.

Code that is common to both Doctor and Patient – in this case, the fields for name and contact number and the method getName - has been extracted from each to make a more abstract superclass called Person. Patient and Doctor inherit from, or extend, Person. As both fields are declared as protected access, the subclasses can access them as if they were their own declared fields.

A computer screen shot of a computer flow chart

Description automatically generated with medium confidence

Here is the code for the three classes. Note: we will assume that there are separate systems for handling bed phones and pagers called **TextSystem** and **PagerSystem**.

**public class Person {  
 protected String name;  
 protected String contactNum;  
  
 public Person(String name) {  
 this.name = name;  
 }  
 public String getName() {  
 return name;  
 }  
}**

**public class Doctor extends Person { //Because the constructor of Person, expects a name!  
 public Doctor(String name) {  
 super(name);  
 }  
 public String getPagerNum() {  
 return contactNum;  
 }  
 public void setPagerNum(String pagerNum) {**

**if (!PagerSystem.checkValid(pagerNum))**

**throw new Exception(“Invalid pager number”);  
 contactNum=pagerNum;  
 }**

**public void contact(String msg) {  
 PagerSystem.send(contactNum,msg);  
 }  
}**

Note: when we instantiate an object of class Patient or Doctor, we are also implicitly instantiating an associated object of class Person.

The keyword ‘**super**’ is the reference to the superclass’s object. The subclasses can use it to specifically access fields and methods in the superclass: eg **super.getName**(); is valid. Calling ‘**super**’ directly calls the superclass’s constructor.

**public class Patient extends Person {**

**private int bedID;  
  
 public Patient(String name) {  
 super(name);  
 }**

**public int getBedID() {  
 return bedID;  
 }  
 public void setBedID(int bedID) {**

**if (bedID>0))**

**this.bedID=bedID;  
 }**

**public String getMobileNum() {  
 return contactNum;  
 }  
 public void setMobileNum(String mobileNum) {**

**if (!TextSystem.checkValid(mobileNum))**

**throw new Exception(“Invalid mobile number”);  
 contactNum=mobileNum;  
 }**

**public void contact(String msg) {  
 TextSystem.send(contactNum,msg);  
 }  
}**

Why have specific methods for setting mobileNum and pagerNumber if the number ends up in the same field? It makes clear to calling code what number needs to be passed in, and the methods in this case have separate code for checking the validity of bed phone numbers and pager numbers.

**Overriding**

**A blue rectangular object with black text

Description automatically generated**

The method ‘getName’ in ICUWard is said to ‘override’ the method ‘getName’ in Ward – ie replace the method when inheriting from Ward, as long as they have the same name and arguments (signature). Ie the following code will call the more specific **ICUWard.getName()** instead of Ward.getName().

**public class Ward {  
 *// Fields* protected String name;  
  
 *// Methods  
 // Constructor* public Ward(String name) {  
 this.name = name;  
 }  
 public String getName() {  
 return name;  
 }  
}**

**public class ICUWard extends Ward{  
  
 public ICUWard(String name) {  
 super(name);  
 }  
 public String getName() {  
 return "ICU Ward:"+name;  
 }  
}**

**ICUWard icu1 = new ICUWard("Hope");  
System.out.println(icu1.getName());**

The overriding method ‘getName()’ in ICUWard *could* if it wanted also call the overridden method in Ward as follows:-

**public String getName(){**

**String s=super.getName();**

**return(“ICU Ward:”+s);  
 }**

**Overloading**

Java can ‘overload’ – ie have several methods with the same name but different arguments. The method that matches the caller’s arguments will be called. Eg:

**public void contact(String message) {PagerSystem.*send*(message);}  
public void contact(String message, String from) {PagerSystem.*send*(message+from);}**

1. **Packages**

**Packages**

Java classes can be organized into logical groups known as packages. A class’s package effectively changes the name of the class by prepending the package name – this helps to distinguish between two different classes having the same name.

For example, we could group all of the .class files for the Doctor, Patient, Person classes into a package called people by adding the following as the first line of each of the corresponding source files:-

**package** people;

The fully qualified name of the Patient class is now people.Patient, so to instantiate the class, we could say:-

people.Patient pat1=**new** people.Patient(“John Smith”);

However, when using classes from a package, we can use **import** to avoid having to prepend the package name. If the following two lines are put at the top of a source file that uses Patient and Doctor classes from the people package, then we can refer to them just as Patient and Doctor within that class:-

**import** people.Patient;

**import** people.Doctor;

So the following is now fine

Patient pat1=**new** Patient(“John Smith”);

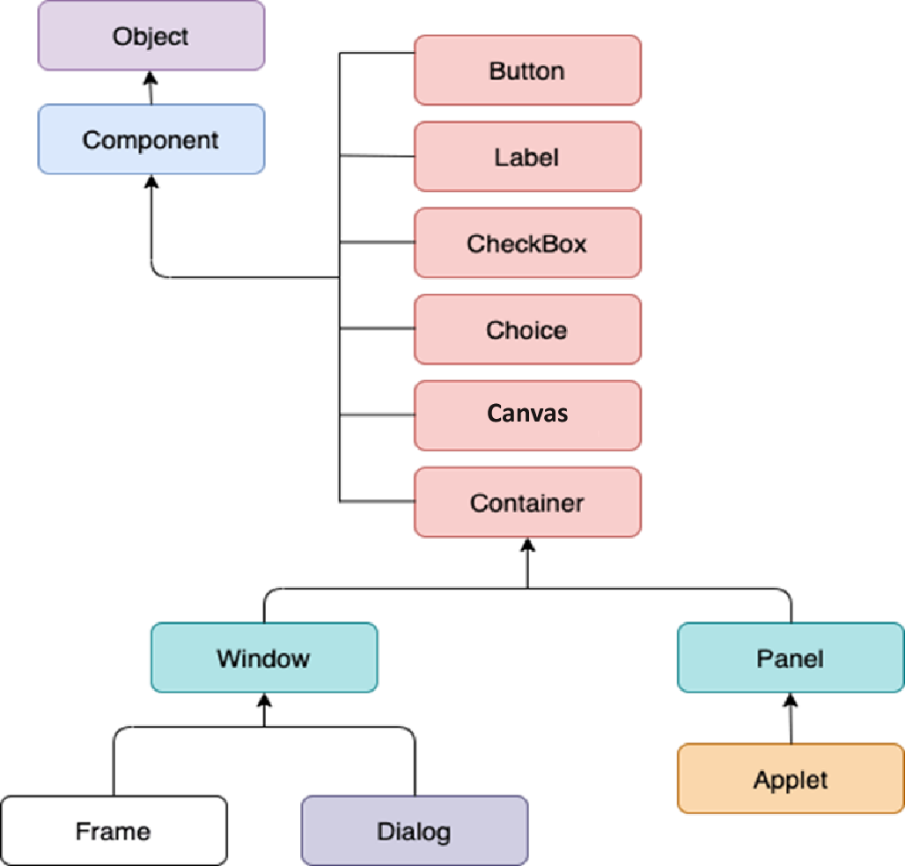
We can import all of the classes in the package with a \*:-

**import** people.\*;

However, if we want to use two different classes from different packages that happen to have the same name we can’t import them both as there would be a clash of class name. We could import one, but would have to use the fully qualified name of the other, or we could use the fully qualified name for both for clarity.

1. **AWT**

Abstract Windows Toolkit (AWT) is a built-in Java library to create Java applications with a basic GUI (graphical user interface). The class hierarchy of AWT includes:-



Ref:JavaTPoint.com

We will be using the Canvas and Frame objects, which are components we can draw shapes on, and a window to host the canvas respectively – we ‘add’ the canvas to the frame.

Canvas has a method called ‘paint’. **We do not call this ourselves**, but it is called by the windows OS whenever it thinks the canvas should be redrawn. It passes into the paint method a ‘Graphics’ object which contains useful drawing methods. If we want to draw on the canvas, we need to make a class that inherits from Canvas and overrides the paint method with our drawing code. In this case the subclass has been called ‘Drawing’.

**A diagram of a square with a frame and text

Description automatically generated with medium confidence**

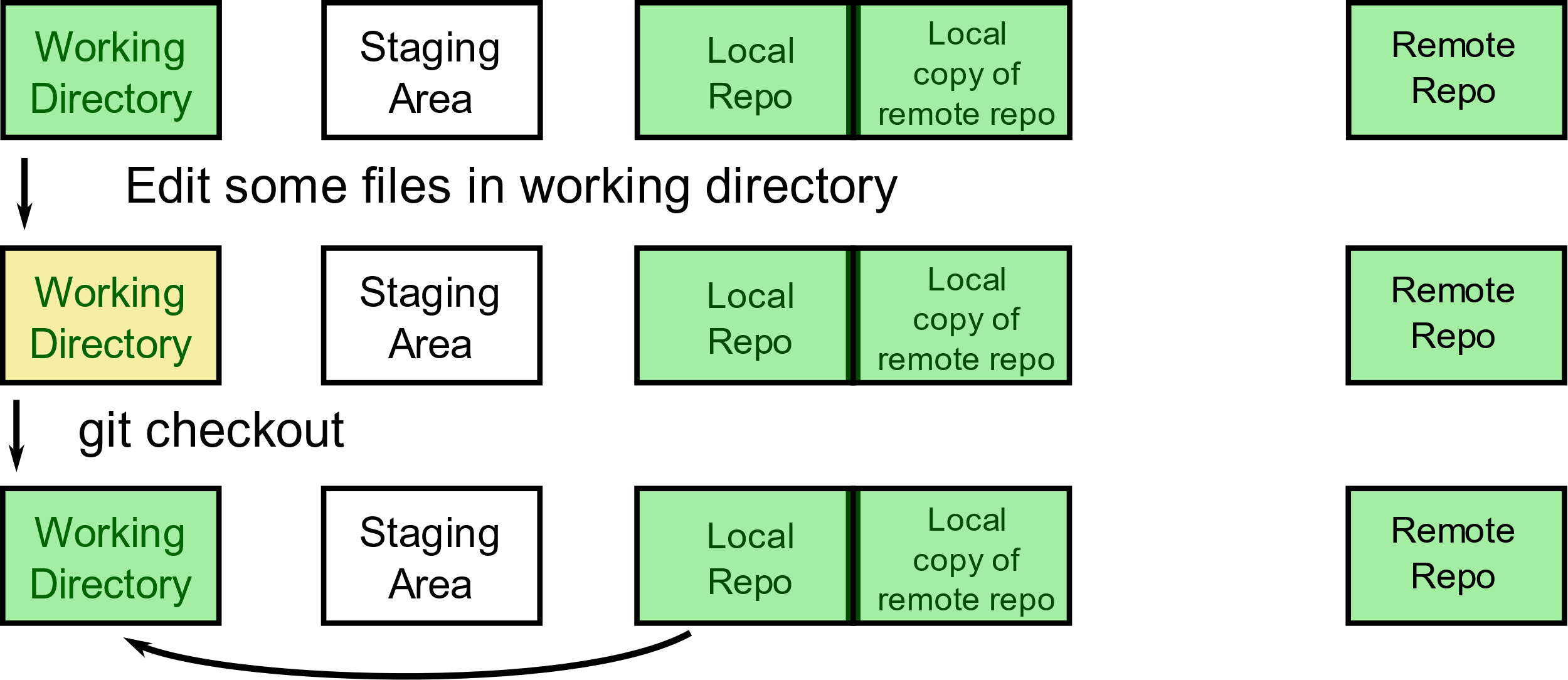
**public class Drawing extends Canvas {  
 public void paint(Graphics g) {  
 g.setColor(Color.*black*);  
 g.drawLine(200, 0, 100, 100);  
 g.setColor(Color.*red*);  
 g.fillOval(300, 0, 50, 100);  
 g.setColor(Color.*blue*);  
 g.fillRect(0, 0, 100, 100);  
 }  
}**

**Frame frame=new Frame("My window");  
 Drawing drawing = new Drawing();  
 frame.add(drawing);**

1. **Git: Code Change Management**

**Undoing Changes**

If the developer was changing code in their working folder, but wants to revert to the code before they started making changes, the git **‘checkout -–force’** command copies the most recently committed files from the local repo back to the working folder, overwriting the changes.

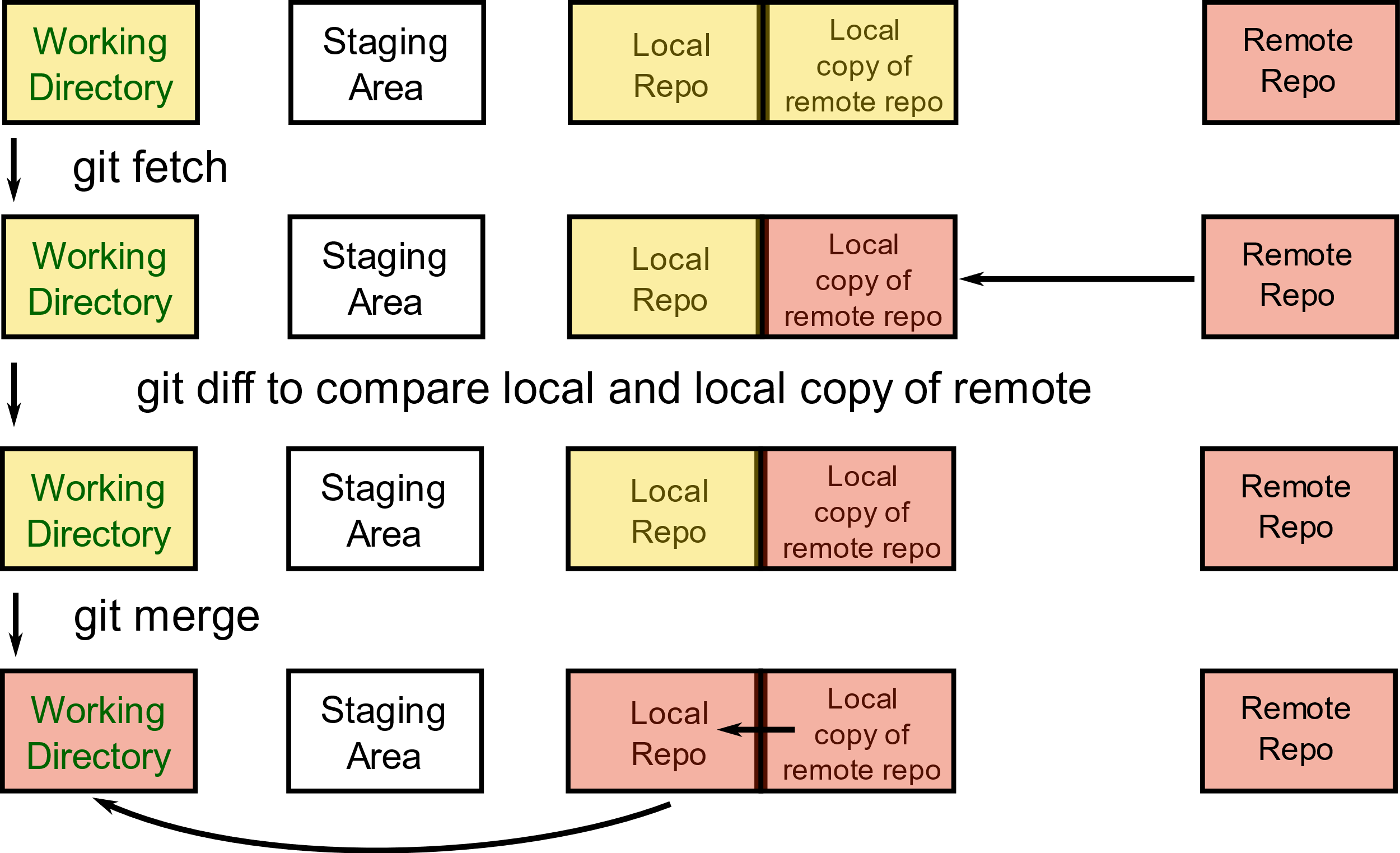


**Comparing the Remote Repo with the Local Repo**

If the developer knows that a new version has been pushed to the remote repo by another person, and they want to compare it with their current version to see what changes have been made, they can use the Git command ‘fetch’. This gets a copy of the latest code in the remote repo and stores it locally, but doesn’t overwrite the local repo. Git command ‘diff’ can then be used to compare the two repos.

The developer can then merge the remote changes into their local repo and working folder with the command ‘merge’ if there are no conflicts. A conflict occurs when the developer has modified a file locally, and the same file has also been modified on the remote repo by someone else. A fetch followed by a merge is the same as a pull.

Get latest version from remote to compare with local, and then merge with local if no conflicts (note git fetch followed by git merge is equivalent to git pull)



**Resolving Conflicts**

If the developer has modified a file in the local repo and someone else has modified the same file in the remote repo, there are conflicts that need resolving. A Git pull will update the local copy of the remote repo with the new files on the remote repo, but will not merge them with the local repo or working folder if conflicts are detected. The developer will then need to edit the source until the conflicts are resolved and then add and commit the changes. The conflicts are indicated in the source files.

