In [2]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, precision_score, recall_score, confusio
from sklearn.preprocessing import LabelEncoder, StandardScaler
import warnings
warnings.filterwarnings('ignore')

plt.style.use('seaborn-v0_8')
sns.set_palette("husl")

print("✓ All libraries imported successfully!")
print("✓ Environment ready for AI Ethics project!")
```

```
✓ All libraries imported successfully!
✓ Environment ready for AI Ethics project!
```

In [3]:
```
def create_synthetic_credit_dataset(n_samples=1000):
    """Create a synthetic credit dataset with potential bias patterns"""

    np.random.seed(42)
    age = np.random.normal(45, 15, n_samples)
    age = np.clip(age, 18, 80)

    age_group = pd.cut(age, bins=[0, 30, 50, 100], labels=['Young', 'Middle', 'Seni

    gender = np.random.choice(['Male', 'Female'], n_samples, p=[0.6, 0.4])

    base_income = np.random.lognormal(10, 0.5, n_samples)
    gender_multiplier = np.where(gender == 'Male', 1.2, 1.0)
    income = base_income * gender_multiplier

    credit_history = np.random.choice(['Good', 'Poor', 'Critical'], n_samples, p=[0

    employment = np.random.choice(['Employed', 'Unemployed', 'Self-employed'], n_sa

    loan_amount = np.random.lognormal(8, 0.8, n_samples)

    approval_prob = 0.3


    approval_prob += np.where(gender == 'Male', 0.15, 0.0)
    approval_prob += np.where(credit_history == 'Good', 0.3, 0.0)
    approval_prob += np.where(credit_history == 'Poor', -0.1, 0.0)
    approval_prob += np.where(credit_history == 'Critical', -0.25, 0.0)
    approval_prob += np.where(income > np.median(income), 0.2, -0.1)
    approval_prob += np.where(employment == 'Employed', 0.1, 0.0)
    approval_prob += np.where(age_group == 'Young', -0.05, 0.0)

    loan_approved = np.random.binomial(1, np.clip(approval_prob, 0, 1), n_samples)
```

```
    df = pd.DataFrame({
        'age': age,
        'age_group': age_group,
        'gender': gender,
        'income': income,
        'credit_history': credit_history,
        'employment_status': employment,
        'loan_amount': loan_amount,
        'loan_approved': loan_approved
    })

    return df

print("✓ Data generation function created")
```

```
✓ Data generation function created
```

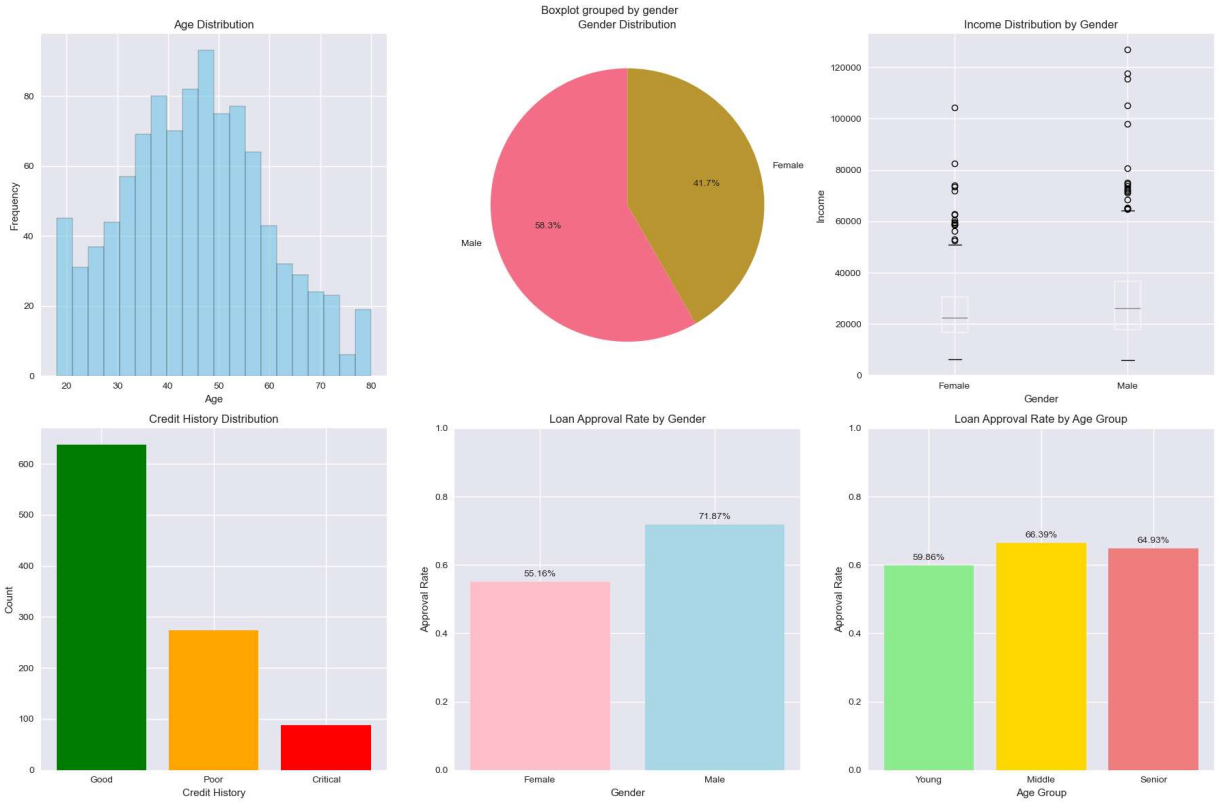
In [4]:
```
df = create_synthetic_credit_dataset(1000)

print("Dataset Overview:")
print(f"Shape: {df.shape}")
print(f"Columns: {list(df.columns)}")
print("\nFirst 5 rows:")
print(df.head())

print("\nDataset Info:")
print(df.info())

print("\nBasic Statistics:")
print(df.describe())

print(f"\nMissing Values:\n{df.isnull().sum()}")

print(f"\nLoan Approval Distribution:")
print(df['loan_approved'].value_counts())
print(f"Approval Rate: {df['loan_approved'].mean():.2%}")
```

```
Dataset Overview:
Shape: (1000, 8)
Columns: ['age', 'age_group', 'gender', 'income', 'credit_history', 'employment_stat
us', 'loan_amount', 'loan_approved']

First 5 rows:
         age age_group  gender        income credit_history employment_status  \
0  52.450712    Senior    Male  22645.513193           Good          Employed
1  42.926035    Middle    Male  18146.688187           Good          Employed
2  54.715328    Senior  Female  25837.630511           Poor          Employed
3  67.845448    Senior  Female  43054.637301           Good          Employed
4  41.487699    Middle    Male  10349.932966           Good          Employed

    loan_amount  loan_approved
0    447.050924              1
1  15270.157650              1
2   3775.257807              1
3   1601.667716              0
4   3380.241161              1

Dataset Info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 8 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   age                1000 non-null   float64
 1   age_group          1000 non-null   category
 2   gender             1000 non-null   object
 3   income             1000 non-null   float64
 4   credit_history     1000 non-null   object
 5   employment_status  1000 non-null   object
 6   loan_amount        1000 non-null   float64
 7   loan_approved      1000 non-null   int32
dtypes: category(1), float64(3), int32(1), object(3)
memory usage: 52.0+ KB
None

Basic Statistics:
               age         income   loan_amount  loan_approved
count  1000.000000    1000.000000   1000.000000    1000.000000
mean     45.376027   27825.887421   4009.046637       0.649000
std      14.193475   14711.864630   3785.066087       0.477522
min      18.000000    5840.463732    234.777193       0.000000
25%      35.286145   17412.118808   1747.633018       0.000000
50%      45.379509   24712.248052   2975.456145       1.000000
75%      54.719158   34768.977118   5050.905757       1.000000
max      80.000000  126905.224571  35965.958068       1.000000

Missing Values:
age                 0
age_group           0
gender              0
income              0
credit_history      0
employment_status   0
```

```
loan_amount          0
loan_approved        0
dtype: int64

Loan Approval Distribution:
loan_approved
1    649
0    351
Name: count, dtype: int64
Approval Rate: 64.90%
```

In [5]:
```
fig, axes = plt.subplots(2, 3, figsize=(18, 12))
fig.suptitle('Dataset Overview and Bias Analysis Dashboard', fontsize=16)

axes[0, 0].hist(df['age'], bins=20, alpha=0.7, color='skyblue', edgecolor='black')
axes[0, 0].set_title('Age Distribution')
axes[0, 0].set_xlabel('Age')
axes[0, 0].set_ylabel('Frequency')

gender_counts = df['gender'].value_counts()
axes[0, 1].pie(gender_counts.values, labels=gender_counts.index, autopct='%1.1f%%',
axes[0, 1].set_title('Gender Distribution')


df.boxplot(column='income', by='gender', ax=axes[0, 2])
axes[0, 2].set_title('Income Distribution by Gender')
axes[0, 2].set_xlabel('Gender')
axes[0, 2].set_ylabel('Income')

credit_counts = df['credit_history'].value_counts()
axes[1, 0].bar(credit_counts.index, credit_counts.values, color=['green', 'orange',
axes[1, 0].set_title('Credit History Distribution')
axes[1, 0].set_xlabel('Credit History')
axes[1, 0].set_ylabel('Count')


approval_by_gender = df.groupby('gender')['loan_approved'].mean()
axes[1, 1].bar(approval_by_gender.index, approval_by_gender.values, color=['pink',
axes[1, 1].set_title('Loan Approval Rate by Gender')
axes[1, 1].set_xlabel('Gender')
axes[1, 1].set_ylabel('Approval Rate')
axes[1, 1].set_ylim(0, 1)


for i, v in enumerate(approval_by_gender.values):
    axes[1, 1].text(i, v + 0.01, f'{v:.2%}', ha='center', va='bottom')

approval_by_age = df.groupby('age_group')['loan_approved'].mean()
axes[1, 2].bar(approval_by_age.index, approval_by_age.values, color=['lightgreen',
axes[1, 2].set_title('Loan Approval Rate by Age Group')
axes[1, 2].set_xlabel('Age Group')
axes[1, 2].set_ylabel('Approval Rate')
axes[1, 2].set_ylim(0, 1)


for i, v in enumerate(approval_by_age.values):
```

```
    axes[1, 2].text(i, v + 0.01, f'{v:.2%}', ha='center', va='bottom')

plt.tight_layout()
plt.show()
```



In [6]:

```
class FairnessMetrics:
    """Class to calculate various fairness metrics"""

    def __init__(self, df, protected_attribute, outcome, prediction):
        self.df = df
        self.protected_attribute = protected_attribute
        self.outcome = outcome
        self.prediction = prediction

    def demographic_parity(self):
        """Calculate demographic parity difference"""
        groups = self.df[self.protected_attribute].unique()
        rates = {}

        for group in groups:
            group_data = self.df[self.df[self.protected_attribute] == group]
            rate = group_data[self.prediction].mean()
            rates[group] = rate

        max_diff = max(rates.values()) - min(rates.values())
        return rates, max_diff

    def equalized_odds_difference(self):
        """Calculate equalized odds difference"""
        groups = self.df[self.protected_attribute].unique()
        tpr_diff = {}
```

```
        fpr_diff = {}

        for group in groups:
            group_data = self.df[self.df[self.protected_attribute] == group]

            tp = len(group_data[(group_data[self.outcome] == 1) & (group_data[self.
            p = len(group_data[group_data[self.outcome] == 1])
            tpr = tp / p if p > 0 else 0

            fp = len(group_data[(group_data[self.outcome] == 0) & (group_data[self.
            n = len(group_data[group_data[self.outcome] == 0])
            fpr = fp / n if n > 0 else 0

            tpr_diff[group] = tpr
            fpr_diff[group] = fpr

        tpr_max_diff = max(tpr_diff.values()) - min(tpr_diff.values())
        fpr_max_diff = max(fpr_diff.values()) - min(fpr_diff.values())

        return tpr_diff, fpr_diff, tpr_max_diff, fpr_max_diff

    def calibration_difference(self):
        """Calculate calibration difference"""
        groups = self.df[self.protected_attribute].unique()
        ppv_rates = {}

        for group in groups:
            group_data = self.df[self.df[self.protected_attribute] == group]

            tp = len(group_data[(group_data[self.outcome] == 1) & (group_data[self.
            pred_pos = len(group_data[group_data[self.prediction] == 1])
            ppv = tp / pred_pos if pred_pos > 0 else 0

            ppv_rates[group] = ppv

        max_diff = max(ppv_rates.values()) - min(ppv_rates.values())
        return ppv_rates, max_diff

print("✓ Fairness Metrics class created")
```

```
✓ Fairness Metrics class created
```

In [7]:

```
def prepare_data(df):
    """Prepare data for machine learning"""
    df_ml = df.copy()

    le_gender = LabelEncoder()
    le_credit = LabelEncoder()
    le_employment = LabelEncoder()
    le_age_group = LabelEncoder()

    df_ml['gender_encoded'] = le_gender.fit_transform(df_ml['gender'])
    df_ml['credit_history_encoded'] = le_credit.fit_transform(df_ml['credit_history
    df_ml['employment_encoded'] = le_employment.fit_transform(df_ml['employment_sta
    df_ml['age_group_encoded'] = le_age_group.fit_transform(df_ml['age_group'])
```

```
    # Select features for model
    features = ['age', 'gender_encoded', 'income', 'credit_history_encoded',
                'employment_encoded', 'loan_amount', 'age_group_encoded']

    X = df_ml[features]
    y = df_ml['loan_approved']

    return X, y, df_ml, {
        'gender': le_gender,
        'credit': le_credit,
        'employment': le_employment,
        'age_group': le_age_group
    }

X, y, df_ml, encoders = prepare_data(df)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

print("✓ Data prepared for machine learning")
print(f"Training set size: {X_train.shape}")
print(f"Test set size: {X_test.shape}")
```

```
✓ Data prepared for machine learning
Training set size: (800, 7)
Test set size: (200, 7)
```

In [8]:

```
print("Training Baseline Model...")
baseline_model = RandomForestClassifier(n_estimators=100, random_state=42)
baseline_model.fit(X_train_scaled, y_train)

y_pred_baseline = baseline_model.predict(X_test_scaled)
y_prob_baseline = baseline_model.predict_proba(X_test_scaled)[:, 1]

baseline_accuracy = accuracy_score(y_test, y_pred_baseline)
baseline_precision = precision_score(y_test, y_pred_baseline)
baseline_recall = recall_score(y_test, y_pred_baseline)

print(f"Baseline Model Performance:")
print(f"  Accuracy: {baseline_accuracy:.3f}")
print(f"  Precision: {baseline_precision:.3f}")
print(f"  Recall: {baseline_recall:.3f}")

df_test = df.iloc[X_test.index].copy()
df_test['prediction'] = y_pred_baseline
df_test['prediction_prob'] = y_prob_baseline

print("✓ Baseline model trained and evaluated")
```

```
Training Baseline Model...
Baseline Model Performance:
  Accuracy: 0.710
  Precision: 0.750
  Recall: 0.831
✓ Baseline model trained and evaluated
```

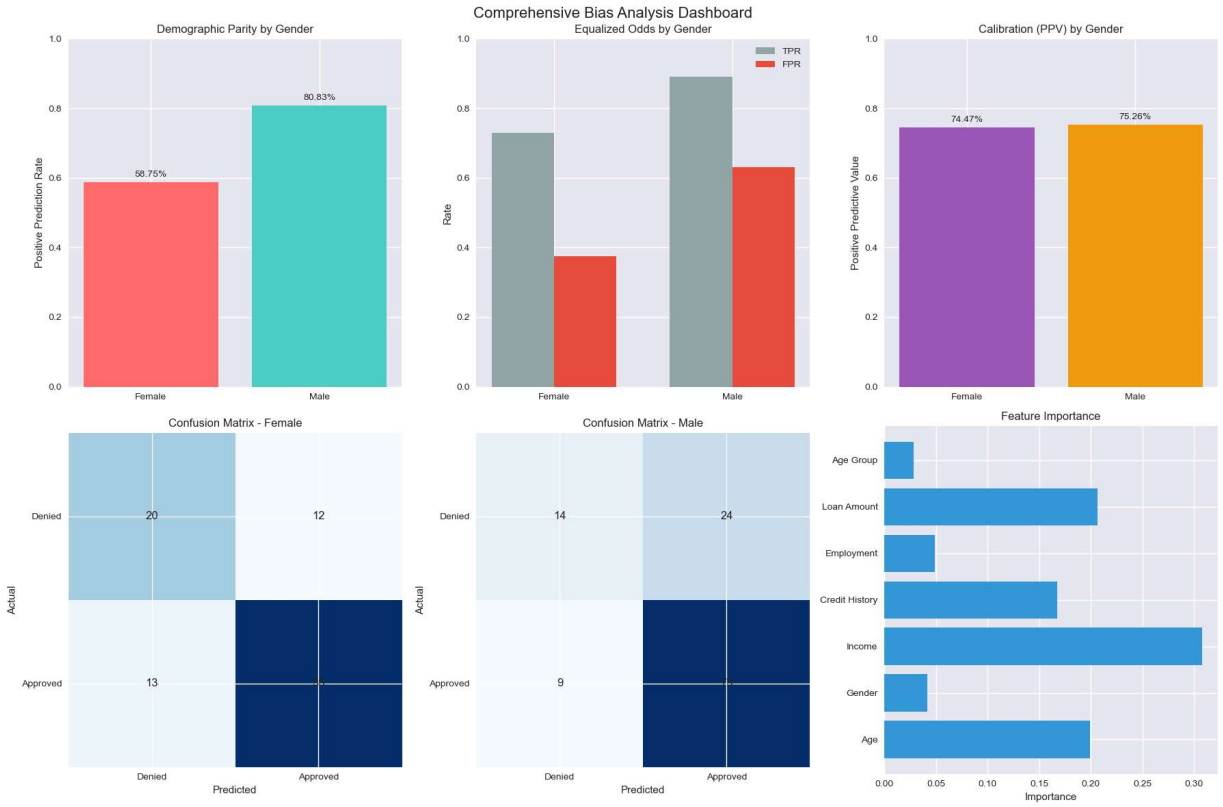
In [9]:
```
print("QUANTITATIVE FAIRNESS METRICS")
print("="*50)

fairness_metrics = FairnessMetrics(df_test, 'gender', 'loan_approved', 'prediction'

dp_rates, dp_diff = fairness_metrics.demographic_parity()
print("Metric 1: Demographic Parity")
print(f"  Rates: {dp_rates}")
print(f"  Difference: {dp_diff:.3f}")

tpr_rates, fpr_rates, tpr_diff, fpr_diff = fairness_metrics.equalized_odds_differen
print("\nMetric 2: Equalized Odds")
print(f"  TPR Rates: {tpr_rates}")
print(f"  FPR Rates: {fpr_rates}")
print(f"  TPR Difference: {tpr_diff:.3f}")
print(f"  FPR Difference: {fpr_diff:.3f}")

cal_rates, cal_diff = fairness_metrics.calibration_difference()
print("\nMetric 3: Calibration")
print(f"  PPV Rates: {cal_rates}")
print(f"  Difference: {cal_diff:.3f}")

print("\n" + "="*50)
print("BIAS INTERPRETATION:")
print("="*50)
print(f"• Demographic Parity Gap: {dp_diff:.1%} - {'HIGH BIAS' if dp_diff > 0.1 els
print(f"• Equalized Odds Gap: {max(tpr_diff, fpr_diff):.1%} - {'HIGH BIAS' if max(t
print(f"• Calibration Gap: {cal_diff:.1%} - {'HIGH BIAS' if cal_diff > 0.1 else 'MO
```

```
QUANTITATIVE FAIRNESS METRICS
==================================================
Metric 1: Demographic Parity
  Rates: {'Female': 0.5875, 'Male': 0.8083333333333333}
  Difference: 0.221

Metric 2: Equalized Odds
  TPR Rates: {'Female': 0.7291666666666666, 'Male': 0.8902439024390244}
  FPR Rates: {'Female': 0.375, 'Male': 0.631578947368421}
  TPR Difference: 0.161
  FPR Difference: 0.257

Metric 3: Calibration
  PPV Rates: {'Female': 0.7446808510638298, 'Male': 0.7525773195876289}
  Difference: 0.008

==================================================
BIAS INTERPRETATION:
==================================================
• Demographic Parity Gap: 22.1% - HIGH BIAS
• Equalized Odds Gap: 25.7% - HIGH BIAS
• Calibration Gap: 0.8% - LOW BIAS
```

In [17]:

```
fig, axes = plt.subplots(2, 3, figsize=(18, 12))
fig.suptitle('Comprehensive Bias Analysis Dashboard', fontsize=16)

groups = list(dp_rates.keys())
rates = list(dp_rates.values())
colors = ['#FF6B6B', '#4ECDC4']

axes[0, 0].bar(groups, rates, color=colors)
axes[0, 0].set_title('Demographic Parity by Gender')
axes[0, 0].set_ylabel('Positive Prediction Rate')
axes[0, 0].set_ylim(0, 1)

for i, v in enumerate(rates):
    axes[0, 0].text(i, v + 0.01, f'{v:.2%}', ha='center', va='bottom')


x_pos = np.arange(len(groups))
width = 0.35

tpr_values = list(tpr_rates.values())
fpr_values = list(fpr_rates.values())

axes[0, 1].bar(x_pos - width/2, tpr_values, width, label='TPR', color='#95A5A6')
axes[0, 1].bar(x_pos + width/2, fpr_values, width, label='FPR', color='#E74C3C')
axes[0, 1].set_title('Equalized Odds by Gender')
axes[0, 1].set_ylabel('Rate')
axes[0, 1].set_xticks(x_pos)
axes[0, 1].set_xticklabels(groups)
axes[0, 1].legend()
axes[0, 1].set_ylim(0, 1)

cal_values = list(cal_rates.values())
axes[0, 2].bar(groups, cal_values, color=['#9B59B6', '#F39C12'])
```

```
axes[0, 2].set_title('Calibration (PPV) by Gender')
axes[0, 2].set_ylabel('Positive Predictive Value')
axes[0, 2].set_ylim(0, 1)

for i, v in enumerate(cal_values):
    axes[0, 2].text(i, v + 0.01, f'{v:.2%}', ha='center', va='bottom')

for i, gender in enumerate(['Female', 'Male']):
    gender_data = df_test[df_test['gender'] == gender]
    cm = confusion_matrix(gender_data['loan_approved'], gender_data['prediction'])

    im = axes[1, i].imshow(cm, interpolation='nearest', cmap='Blues')
    axes[1, i].set_title(f'Confusion Matrix - {gender}')


    for j in range(cm.shape[0]):
        for k in range(cm.shape[1]):
            axes[1, i].text(k, j, str(cm[j, k]), ha='center', va='center', fontsize

    axes[1, i].set_xlabel('Predicted')
    axes[1, i].set_ylabel('Actual')
    axes[1, i].set_xticks([0, 1])
    axes[1, i].set_yticks([0, 1])
    axes[1, i].set_xticklabels(['Denied', 'Approved'])
    axes[1, i].set_yticklabels(['Denied', 'Approved'])

feature_importance = baseline_model.feature_importances_
feature_names = ['Age', 'Gender', 'Income', 'Credit History', 'Employment', 'Loan A

axes[1, 2].barh(feature_names, feature_importance, color='#3498DB')
axes[1, 2].set_title('Feature Importance')
axes[1, 2].set_xlabel('Importance')

plt.tight_layout()
plt.show()
```



In [11]:
```
print("BIAS MITIGATION TECHNIQUE 1: REWEIGHTING")
print("="*50)

def calculate_reweighting_factors(df, protected_attribute, outcome):
    """Calculate reweighting factors for bias mitigation"""

    overall_positive_rate = df[outcome].mean()

    weights = {}
    for group in df[protected_attribute].unique():
        group_data = df[df[protected_attribute] == group]
        group_positive_rate = group_data[outcome].mean()
        group_size = len(group_data)

        if group_positive_rate > 0:
            weight_positive = overall_positive_rate / group_positive_rate
        else:
            weight_positive = 1.0

        weight_negative = (1 - overall_positive_rate) / (1 - group_positive_rate) i

        weights[group] = {
            'positive': weight_positive,
            'negative': weight_negative,
            'size': group_size
        }

    return weights

reweight_factors = calculate_reweighting_factors(df, 'gender', 'loan_approved')
print("Reweighting Factors:")
```

```
for group, factors in reweight_factors.items():
    print(f"  {group}: Positive={factors['positive']:.3f}, Negative={factors['negat

def apply_reweighting(X_train, y_train, df_train, protected_attribute, reweight_fac
    """Apply reweighting to training data"""
    sample_weights = np.ones(len(X_train))

    for i, (idx, row) in enumerate(df_train.iterrows()):
        group = row[protected_attribute]
        outcome = row['loan_approved']

        if outcome == 1:
            sample_weights[i] = reweight_factors[group]['positive']
        else:
            sample_weights[i] = reweight_factors[group]['negative']

    return sample_weights

df_train = df.iloc[X_train.index].copy()
sample_weights = apply_reweighting(X_train, y_train, df_train, 'gender', reweight_f

print("Training Reweighted Model...")
reweighted_model = RandomForestClassifier(n_estimators=100, random_state=42)
reweighted_model.fit(X_train_scaled, y_train, sample_weight=sample_weights)

y_pred_reweighted = reweighted_model.predict(X_test_scaled)
df_test['prediction_reweighted'] = y_pred_reweighted

fairness_reweighted = FairnessMetrics(df_test, 'gender', 'loan_approved', 'predicti
dp_rates_rw, dp_diff_rw = fairness_reweighted.demographic_parity()

print(f"Reweighted Model - Demographic Parity:")
print(f"  Rates: {dp_rates_rw}")
print(f"  Difference: {dp_diff_rw:.3f} (vs {dp_diff:.3f} baseline)")
print(f"  Improvement: {((dp_diff - dp_diff_rw) / dp_diff * 100):.1f}%")
```

```
BIAS MITIGATION TECHNIQUE 1: REWEIGHTING
==================================================
Reweighting Factors:
  Male: Positive=0.903, Negative=1.248
  Female: Positive=1.177, Negative=0.783
Training Reweighted Model...
Reweighted Model - Demographic Parity:
  Rates: {'Female': 0.6, 'Male': 0.7833333333333333}
  Difference: 0.183 (vs 0.221 baseline)
  Improvement: 17.0%
```

In [13]:

```
print("BIAS MITIGATION TECHNIQUE 2: THRESHOLD OPTIMIZATION")
print("="*50)

def optimize_thresholds(df, protected_attribute, outcome, prediction_prob):
    """Optimize decision thresholds for different groups"""
    groups = df[protected_attribute].unique()
    optimal_thresholds = {}

    for group in groups:
```

```
        group_data = df[df[protected_attribute] == group]

        best_threshold = 0.5
        best_accuracy = 0

        for threshold in np.arange(0.1, 0.9, 0.05):
            pred = (group_data[prediction_prob] >= threshold).astype(int)
            accuracy = accuracy_score(group_data[outcome], pred)

            if accuracy > best_accuracy:
                best_accuracy = accuracy
                best_threshold = threshold

        optimal_thresholds[group] = best_threshold

    return optimal_thresholds

optimal_thresholds = optimize_thresholds(df_test, 'gender', 'loan_approved', 'predi
print("Optimal Thresholds:")
for group, threshold in optimal_thresholds.items():
    print(f"  {group}: {threshold:.3f}")

def apply_optimized_thresholds(df, protected_attribute, prediction_prob, thresholds
    """Apply group-specific thresholds"""
    predictions = np.zeros(len(df))

    for group, threshold in thresholds.items():
        group_mask = df[protected_attribute] == group
        predictions[group_mask] = (df.loc[group_mask, prediction_prob] >= threshold

    return predictions

y_pred_optimized = apply_optimized_thresholds(df_test, 'gender', 'prediction_prob',
df_test['prediction_optimized'] = y_pred_optimized

fairness_optimized = FairnessMetrics(df_test, 'gender', 'loan_approved', 'predictio
dp_rates_opt, dp_diff_opt = fairness_optimized.demographic_parity()

print(f"Optimized Thresholds - Demographic Parity:")
print(f"  Rates: {dp_rates_opt}")
print(f"  Difference: {dp_diff_opt:.3f} (vs {dp_diff:.3f} baseline)")
print(f"  Improvement: {((dp_diff - dp_diff_opt) / dp_diff * 100):.1f}%")
```

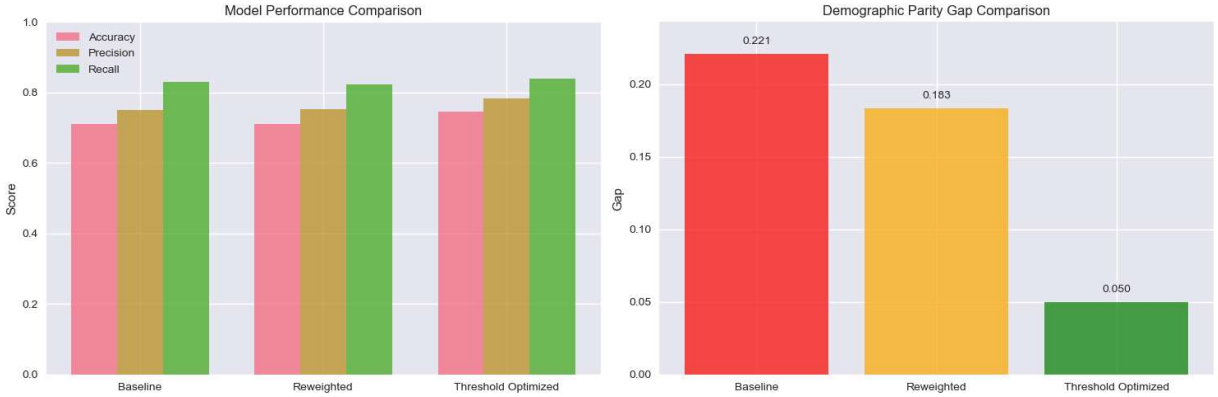
```
BIAS MITIGATION TECHNIQUE 2: THRESHOLD OPTIMIZATION
==================================================
Optimal Thresholds:
  Female: 0.350
  Male: 0.700
Optimized Thresholds - Demographic Parity:
  Rates: {'Female': 0.725, 'Male': 0.675}
  Difference: 0.050 (vs 0.221 baseline)
  Improvement: 77.4%
```

In [14]:
```
print("PERFORMANCE COMPARISON")
print("="*50)
```

```
models = {
    'Baseline': y_pred_baseline,
    'Reweighted': y_pred_reweighted,
    'Threshold Optimized': y_pred_optimized
}

results_summary = []

for model_name, predictions in models.items():

    accuracy = accuracy_score(y_test, predictions)
    precision = precision_score(y_test, predictions)
    recall = recall_score(y_test, predictions)

    if model_name == 'Baseline':
        dp_rates_temp, dp_diff_temp = dp_rates, dp_diff
    elif model_name == 'Reweighted':
        dp_rates_temp, dp_diff_temp = dp_rates_rw, dp_diff_rw
    else:
        dp_rates_temp, dp_diff_temp = dp_rates_opt, dp_diff_opt

    results_summary.append({
        'Model': model_name,
        'Accuracy': accuracy,
        'Precision': precision,
        'Recall': recall,
        'Demographic Parity Gap': dp_diff_temp,
        'Female Approval Rate': dp_rates_temp['Female'],
        'Male Approval Rate': dp_rates_temp['Male']
    })

results_df = pd.DataFrame(results_summary)
print(results_df.round(3))

fig, axes = plt.subplots(1, 2, figsize=(15, 5))

models_list = results_df['Model'].tolist()
x_pos = np.arange(len(models_list))
width = 0.25

axes[0].bar(x_pos - width, results_df['Accuracy'], width, label='Accuracy', alpha=0
axes[0].bar(x_pos, results_df['Precision'], width, label='Precision', alpha=0.8)
axes[0].bar(x_pos + width, results_df['Recall'], width, label='Recall', alpha=0.8)
axes[0].set_title('Model Performance Comparison')
axes[0].set_ylabel('Score')
axes[0].set_xticks(x_pos)
axes[0].set_xticklabels(models_list)
axes[0].legend()
axes[0].set_ylim(0, 1)

axes[1].bar(models_list, results_df['Demographic Parity Gap'], color=['red', 'orang
axes[1].set_title('Demographic Parity Gap Comparison')
axes[1].set_ylabel('Gap')
axes[1].set_ylim(0, max(results_df['Demographic Parity Gap']) * 1.1)
```

```
for i, v in enumerate(results_df['Demographic Parity Gap']):
    axes[1].text(i, v + 0.005, f'{v:.3f}', ha='center', va='bottom')

plt.tight_layout()
plt.show()
```

```
PERFORMANCE COMPARISON
==================================================
                Model  Accuracy  Precision  Recall  Demographic Parity Gap  \
0            Baseline     0.710      0.750   0.831                   0.221
1         Reweighted      0.710      0.754   0.823                   0.183
2  Threshold Optimized     0.745      0.784   0.838                   0.050

   Female Approval Rate  Male Approval Rate
0                 0.588               0.808
1                 0.600               0.783
2                 0.725               0.675
```



In [16]:
```
print("REAL-WORLD IMPLICATIONS & ETHICAL ANALYSIS")
print("="*60)

print("1")
```

```
REAL-WORLD IMPLICATIONS & ETHICAL ANALYSIS
============================================================
1
```

In [ ]: